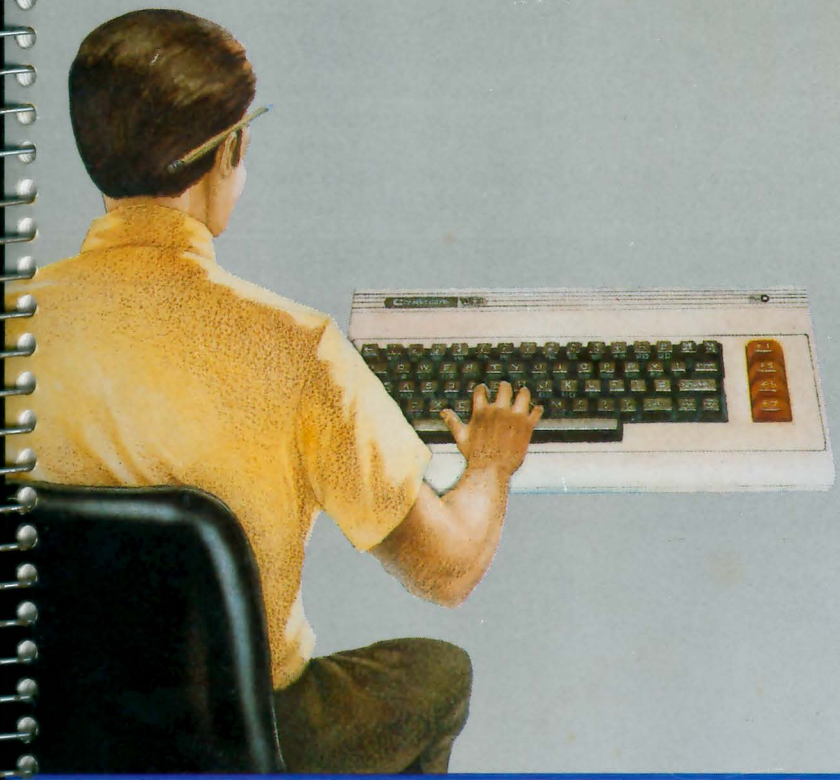
Waterloo Structured Basic for the VIC-20



commodore COMPUTER

Chapter 1

Introduction To Waterloo BASIC

This chapter will serve as an introduction to programming using the Waterloo BASIC system as implemented on Commodore microcomputers. Simple examples are used to illustrate the basic principles. The programming novice will be presented with an overview of the terminology; the experienced programmer will see examples of how familiar concepts are implemented in Waterloo BASIC. The various features discussed are more completely explained in subsequent chapters, and in the Appendices.

1.1 Basic Principles

BASIC is a computer programming language developed initially at Dartmouth College in New Hampshire. It has been widely used, and many dialects of the language have appeared over the years. Waterloo BASIC is the particular version of the language developed on a number of computers at the University of Waterloo.

Example 1.1 illustrates how you could write a BASIC program to calculate the Fahrenheit equivalent of 100 degrees Celsius. This is too trivial a problem to be considered for solution using a computer, but nevertheless, it serves to illustrate a number of important points.

```
1 REM EXAMPLE 1.1
2 REM A CALCULATION
3 C = 100
4 F = (C*9)/5 + 32
5 PRINT C,F
6 STOP
```

This example consists of six *statements*, each preceded by a *line number* or *statement number*. The first two statements are *comments* which describe the problem. These comment statements are recognizable by the REM (remark) which appears immediately following the statement number. Comments are used to identify and describe the program, and are used only for documentation purposes; they are not executed by the computer.

The last four statements are *executable* statements, and will be processed by the computer in the order in which they appear. Statement 3 causes the value 100 to be assigned to the *variable* C. Then statement 4 causes the *expression*

```
(C*9)/5 + 32
```

to be computed, with its value being assigned to F. Note that this expression is similar to expressions used in algebra, with the * used to denote multiplication and the / used to denote division.

Since this is the formula used to convert Celsius to Fahrenheit, F would at this point have the value 212, which is the boiling point of water. The fifth statement will cause the current values of C and F to be printed or displayed on the screen, and statement six causes the computer to terminate execution of the program. These statements are therefore executed *sequentially* by the computer, and the entire collection of statements is referred to as a *program*.

1.2 Line Numbers

It is not necessary to number the statements from 1 to 6, as shown in Example 1.1. In fact, it is more desirable to number the statements leaving *gaps* between the statement numbers in case it should be required to insert a new statement between two existing statements. Thus a more common form of the same program appears as Example 1.2.

```
10 REM EXAMPLE 1.2
20 REM A CALCULATION
30 C = 100
40 F = (C*9)/5 + 32
50 PRINT C,F
60 STOP
```


1.3 Looping

It would be more useful and interesting to calculate F for C having a number of values, for example 100, 110, 120, ... etc. This is accomplished in Example 1.3 by using a *loop*.

```
10 REM EXAMPLE 1.3
15 :
20 C = 100
22 :
25 LOOP
30   F = (C*9)/5 + 32
40   PRINT C,F
45   C = C + 10
47 ENDLOOP
50 STOP
```

Two new statements have been used, namely LOOP and ENDLOOP, to determine the beginning and ending of a group of statements which are to be executed repeatedly by the computer. Thus when Example 1.3 is executed, the statements

```
30      F = (C*9)/5 + 32
40      PRINT C,F
45      C = C + 10
```

are repeated indefinitely. Consequently the computer never stops, and the program is said to be in an *infinite loop*, a most undesirable state of affairs. A method for overcoming this short-coming is illustrated in the next example.

Notes:

(1) Each time through the loop, the value of C is *incremented* by 10 in the statement

```
C = C + 10
```

(2) The = in BASIC means "is assigned the value". Thus this statement is not an algebraic equation, but means "calculate the expression C+10 and assign its value to C". By incrementing C each time through the loop, a table of values of F for C

starting from 100 and increasing in steps of 10 degrees, is produced.

(3) The reader may have noticed that all statements in the *range* of the loop have been indented. This is to make the program more readable, and has no effect on its operation within the computer.

(4) Until we cover the material in Chapter 4, the reader should assume that *variable names* are a single letter, namely A to Z inclusive.

(5) Some of the statements contain only a colon (:). The colon is used to provide a line of spaces between statements so that the programs are easier to read. BASIC does not allow completely blank lines in a program, but the colon acts as a blank line since it invokes no action from BASIC. Lines which only contain colons are called *null statements.*

1.4 Terminating The Loop

```
10 REM EXAMPLE 1.4
18 :
20 C = 100
22 :
25 LOOP
30   F = (C*9)/5 + 32
40   PRINT C,F
45   C = C + 10
46   IF C > 200 THEN QUIT
47 ENDLOOP
48 :
50 STOP
```

Example 1.4 shows how simple it is to cause the computer to *terminate* the loop at a desired value of C. A single statement

```
IF C > 200 THEN QUIT
```

has been inserted in the loop, and when C is incremented to the value 210 the computer will exit from the loop, and continue to the statement immediately following the ENDLOOP statement. In this case that statement is STOP, so the process is terminated.

The quantity C > 200 is known as a *relational expression*, and when computed has the value "true" or "false". The operator > is known as a relational operator. There are six such operators available, namely

=	equals
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
<>	not equal to

1.5 Keywords

In this chapter, several BASIC *keywords* have been introduced. These include REM, PRINT, STOP, LOOP, ENDLOOP, IF, THEN and QUIT. These are reserved words which cannot be used for any other purpose in BASIC. There are dozens of such keywords in BASIC; a complete list is given in the Appendices.

1.6 Spacing Within A Statement

In the various examples blank characters have been used extensively to improve readability. For example, line 20 in Example 1.3 could be written in various ways as follows:

```
20 C = 100
20 C=100
20    C   = 100
20C=100
```

All of these are equivalent. The blank characters are used at the programmer's discretion, and are ignored by the computer.

1.7 Summary

This chapter has introduced many programming terms, and illustrated their application in the BASIC language. No attempt has been made to rigorously define the terms. It is suggested that the reader should proceed immediately to Chapter 2 to learn how to submit these examples to the

computer, and run them to observe their operation. The definitions of terms used are covered more rigorously in the Appendices, and in subsequent chapters.

1.8 Exercises

1.1 Make the required changes to Example 1.4 to produce a table of Fahrenheit values for Celsius values ranging from 0 to 100 degrees in increments of one degree.

1.2 Further change the program to produce only those Fahrenheit values for Celsius values ranging from -5 to 20 degrees in increments of 2 degrees.

1.3 Further change each of the programs to produce the tables in reverse order. That is, the highest temperatures should be computed and printed first, with the lowest at the end of the table. (The subtraction operator in BASIC is the symbol '-' as in algebra).

1.4 Write a program to produce a table showing Fahrenheit to Celsius conversion values.

1.5 With the switch to metric it is important to be able to compare English units and metric units.

a) Write a program which converts 100 miles to kilometers. Use the conversion factor one kilometer equals 5/8 of a mile.

b) It is necessary to produce a speedometer with both miles per hour and kilometers per hour showing on its face in increments of 10 miles per hour and 10 kilometers per hour. Write a program which produces two lists, the first one showing miles per hour from 0 to 100 in 10 miles per hour increments and the corresponding kilometers per hour values, and the second one showing kilometers per hour from 0 to 160 in 10 kilometers per hour increments and the corresponding miles per hour values.

Chapter 2

Running a BASIC Program

The basic principles of writing a simple BASIC program were described in Chapter 1. The next step is to enter the program into the computer, and then put it into operation. This is accomplished using the BASIC editor and run-time systems. In this chapter many of the features of these two systems are discussed.

2.1 Signing on to Waterloo BASIC

Waterloo BASIC is implemented as an addition to the existing BASIC language in the Commodore microcomputer. The method of activating these extensions differs from machine to machine.

In the case of the VIC-20, simply insert the provided cartridge as described in the installation guide and turn on the machine. A message similar to the following will be displayed.

```
*** WATERLOO BASIC ***
*      FOR VIC-20      *
*                      *
*   COPYRIGHT 1983     *
* WATERLOO COMPUTING   *
*** SYSTEMS LIMITED***

6655 BYTES FREE

READY
```

In the case of the Commodore 2000, 4000 or 8000 series computers, you will have to install an EPROM chip by following the directions which were

provided when you obtained Waterloo BASIC. When the chip is properly installed and the machine turned on, the computer will display a message similar to the following:

```
### COMMODORE BASIC ###
 31743 BYTES FREE
READY
```

At this time, Waterloo BASIC must be activated by typing SYS 9*4096 and pressing the RETURN key. The computer will respond with

```
*** WATERLOO STRUCTURED BASIC ***
    COPYRIGHT U OF W; (27/4/80)

READY
```

At this time Waterloo BASIC is active.

Notes:

(1) The above procedures are typical but may vary slightly from machine to machine. If you suspect a difference, consult the installation procedures provided with the Waterloo BASIC package.

(2) Some Commodore computers operate in lower case mode. If you have one of these, the programs in this text should be entered *entirely* in lower case.

(3) If at anytime you wish to disable the Waterloo BASIC features, simply remove the cartridge (in the case of the VIC-20) or type SYS 9*4096+3 (in the case of the 2000, 4000 or 8000 series machines).

2.2 Typing in a BASIC Program

When you first sign on to Waterloo BASIC you are provided with a *workspace* which is empty. You can then enter each line of the program by typing it, together with its statement number, and then depressing the RETURN key. If you make a mistake simply type the line again and the computer system will replace the old line with the new line, provided they have the same statement number.

If you are using a Commodore computer with a 40-column screen or even a 22-column screen, you may find that the line you wish to enter is too long to fit on the screen. In these cases, it can "wrap-around" onto the next line on the screen and still be treated as a single line in the program. For instance, Example 1.4 will appear as follows when entered into the VIC-20 computer.

```
10 REM EXAMPLE 1.4
18 :
20 C=100
22 :
25 LOOP
30   F=(C*9)/5+32
40   PRINT C,F
45   C=C+10
46   IF C > 200 THEN Q
UIT
47 ENDLOOP
48 :
50 STOP
```

Notice how program line number 46 is displayed as two lines on the screen. It is, however, still treated as a single program line by BASIC. Just remember when typing in a program to hit the RETURN key once at the end of every "program" line even if the cursor wraps around to the next screen line.

If you wish to check your typing, enter the *editor* command

```
LIST
```

and the contents of the entire workspace will appear on the screen.

The LIST command can be used to display a "range" of lines on the screen. For example

```
LIST 100-200
```

will cause all lines between 100 and 200 inclusively to be displayed. The command

```
LIST -200
```

will cause all lines up to and including line 200 to be displayed. The command

```
LIST 200-
```

will cause all lines numbered 200 or greater to be displayed.

2.3 Running a BASIC Program

After you have entered the program you are ready to place it into *execution*. This is done by typing RUN and then depressing the RETURN key.

The computer immediately starts executing the BASIC statements in your workspace, beginning with the first executable statement (the statement with the lowest statement number). Execution continues from statement to statement until the STOP statement is encountered, at which time the message

```
BREAK IN "statement number"
```

appears, and the screen once again displays READY status.

The "statement number" indicates the statement number of the STOP statement in the program which was executed.

All too often, an error occurs during execution (usually referred to as "execution time"). For example, if you try to compute the square root of a negative number, the computer must terminate execution because it is unable to perform this function. When this happens an error message appears on the screen. Hopefully the message is self-explanatory, but you may have to refer to the Appendices for further explanation. When the reason for the execution-time error has been determined, simply enter the correction into your workspace, thus *modifying* your original program, and RUN it again.

Occasionally the computer will remain in execution endlessly. It is then said to be in an "infinite loop". When this occurs you must use the STOP key to *interrupt* operation of the program. This returns the computer to the READY status, and indicates the line number which was about to be executed (BREAK message).

2.4 Output from a BASIC Program

When Example 1.4 is run on the computer, the output on the screen appears as follows:

```
100         212
110         230
120         248
130         266
140         284
150         302
160         320
170         338
180         356
190         374
200         392
```

On each line the numbers are arranged in 10-position *zones* or *windows*. This means the first number on any line appears in columns 1 to 10 inclusive, the second number in columns 11 to 20 inclusive, etc. Each number appears *left-justified* in a window, with the sign in the left-most position. When the sign is positive, this position is left blank.

While the Commodore 8032 allows up to 8 of these windows on a line, the PET allows only 4 and the VIC-20 only 2. If a program prints into more windows than will fit on a screen, the output will simply "wrap-around" onto the next screen line.

2.5 Saving a BASIC Program

When you have completed your program you may wish to *SAVE* it on a cassette. First, insert a blank cassette into the cassette unit and make sure it is rewound to the start. Next, think of a name for it such as PROGONE. Then type

```
SAVE "PROGONE"
```

and depress the RETURN key. The following message will appear on the screen:

```
PRESS PLAY & RECORD ON TAPE
```

Press PLAY and RECORD simultaneously on the cassette unit and the computer will respond with OK on the screen, followed by the message

```
WRITING PROGONE
```

The complete program in the workspace will be written onto the cassette tape. When writing is complete, the computer responds with the READY message.

Occasionally the program will be recorded incorrectly, possibly because of a faulty tape or because you forgot to press RECORD along with the PLAY button. It is good practice to VERIFY that it is properly recorded by using the following procedure:

(i) Rewind the cassette tape.

(ii) Type VERIFY "PROGONE". The computer will instruct you to press PLAY. The tape will be read and verified character-by-character to ensure it matches the content of the workspace.

(iii) When the verification process is complete, the computer returns to READY status. If the verification process fails, rewind the tape and try SAVEing the program again, followed by another VERIFY. If the error persists try another tape.

The name you choose for your program can contain up to 16 letters or digits and should begin with a letter. Actually, your program is stored as a *file*, and the name you choose is referred to as the *file name*.

Although you have SAVEd your program, a copy remains in your workspace as well. If you wish to erase it from your workspace you should enter the command

```
NEW
```

Then the workspace will be returned to its original status, as if you had just signed on.

2.6 Loading a BASIC Program from Cassette

At some point in time you will probably want to copy a selected program from your cassette into your workspace. First, insert the desired cassette into the cassette unit and make sure it is rewound to the start. With the system in READY status, simply type

```
LOAD "PROGONE"
```

followed by RETURN and the workspace is automatically cleared. The message

```
PRESS PLAY ON TAPE
```

will be displayed on the screen. Press PLAY and the program will be read into your workspace.

NOTE: Several programs can be stored on one cassette. They are written one after another with about 10 seconds of blank tape separating them. Each program has a "header" which contains its name. If your cassette already contains a program and you wish to record another, it is your responsibility to "fast forward" the tape to a position following the recorded program before SAVEing the new one. This is obviously difficult to do. The only reasonable way to position the tape is by refraining from rewinding it after the recorded program has been SAVEd, LOADed or VERIFYed.

When LOADing a program from a cassette which contains several programs, the proper program is automatically selected using the program name contained in the header. However, if the programs are named PROGONE and PROG, in that order, and you attempt to LOAD "PROG" you will actually get PROGONE. This is because the computer does not actually search for PROG, but for a program whose name *begins* with the sequence PROG as specified in the LOAD command.

2.7 Changing a Program

It has been explained that any line in your workspace can be changed by re-entering it with the same statement number. A line can be completely erased by typing only the line number, followed by RETURN. Lines can also be changed using the full screen editor which is described in an Appendix.

2.8 Summary

A number of commands such as LOAD, SAVE, VERIFY, RUN, NEW, and LIST have been introduced in this chapter. These commands are *editor commands* or *system commands* because they are associated with the editing, managing and operation of the program.

2.9 Exercises:

2.1 Enter each of the examples from Chapter 1 into the computer and run them to ensure that the output is as expected. Store these programs onto a cassette for later reference.

2.2 Enter solutions for the Exercises at the end of Chapter 1 and run the programs to be sure they work.

2.3 Try every editor or system command introduced in this chapter to verify that they work as described.

Chapter 3

The Flavour of Waterloo BASIC

In Chapter 1 simple programs were discussed, and presumably the reader has entered these examples into the computer, run them, and modified them in various ways. In this chapter further examples are used to introduce the reader to additional features of the BASIC language. The purpose of this chapter is to give the reader a feeling for the "flavour" of the system, and to allow more interesting problems to be attempted before examining all of the details. Subsequent chapters repeat most of the material covered, and provide more formal definitions and descriptions.

3.1 Making Headings in BASIC

```
10 REM EXAMPLE 3.1
13 :
15 PRINT "CELSIUS","FAHRENHEIT"
18 :
20 C = 100
22 :
25 LOOP
30   F = (C*9)/5 + 32
40   PRINT C,F
45   C = C + 10
46   IF C > 200 THEN QUIT
47 ENDLOOP
48 :
50 STOP
```

Example 3.1 is identical to Example 1.4, except that one additional statement has been added, namely

```
PRINT "CELSIUS","FAHRENHEIT"
```

When the computer runs the program this statement is encountered before the loop begins, and the words CELSIUS and FAHRENHEIT are printed in the first two windows of the first line. These quantities are referred to as *string constants* and can be recognized because they are enclosed between quotation marks. Such string constants can be printed and used as headings or special text in any window of any line.

The reader should type the program into the system, run it and confirm that the output appears as follows:

```
CELSIUS  FAHRENHEIT
 100      212
 110      230
 120      248
 130      266
 140      284
 150      302
 160      320
 170      338
 180      356
 190      374
 200      392
```

Notice that the words CELSIUS and FAHRENHEIT are printed left-justified in the windows. Actually the numbers are also left-justified but the "+" sign is not printed.

Example 3.2 shows another interesting use for string constants.

```
10 REM EXAMPLE 3.2
15 :
20 C = 100
22 :
25 LOOP
30   F = (C*9)/5 + 32
40   PRINT "C =",C,"F =",F
45   C = C + 10
46   IF C > 200 THEN QUIT
47 ENDLOOP
48 :
50 STOP
```

Here the PRINT statement contains quantities to be printed in four windows on every line. The first and third quantities are string constants, while the second and fourth are variables. When the program is placed into execution the output appears as follows:

```
C =   100   F =   212
C =   110   F =   230
C =   120   F =   248
C =   130   F =   266
C =   140   F =   284
C =   150   F =   302
C =   160   F =   320
C =   170   F =   338
C =   180   F =   356
C =   190   F =   374
C =   200   F =   392
```

NOTE: The output of Example 3.2 contains a considerable number of blank spaces in each line because each of the windows is not completely used.

In fact, on the narrow screen of the VIC-20, it actually prints 2 lines on the screen for each "logical" line of output shown, like this:

```
C =          100
F =          212
C =          110
F =          230
C =          120
F =          248
C =          130
F =          266
C =          140
F =          284
C =          150
F =          302
C =          160
F =          320
C =          170
F =          338
C =          180
F =          356
C =          190
F =          374
C =          200
F =          392
```

This problem can be overcome by using the following alternative for line 40.

```
40 PRINT "C =";C,"F =";F
```

The only change is that two of the commas have been replaced by semicolons. When two items in the "PRINT list" are separated by a semicolon, their values are *concatenated*, with no intervening spaces, and are printed in the next available window as a single unit. In the example, the string "C =" is three characters long; the value of C is 4 digits (including allowance for a sign). Thus the concatenated data is seven characters long, and will print in the first window. Similarly "F =" is concatenated to the value of F to form a seven character result which is printed in the second window. The reader should make this change to Example 3.2 and observe the output.

3.2 More Examples of Algebraic Expressions

a) In previous examples an expression was calculated, namely

```
(C*9)/5 + 32
```

It was noted that this expression is similar to expressions used in algebra. The corresponding expression for conversion of Fahrenheit to Celsius would be as follows:

```
((F - 32)*5)/9
```

Here the subtraction operator - has been introduced, as well as two sets of parentheses which are *nested*. First the computer will calculate the innermost expression, namely F-32. This quantity will be multiplied by 5, and finally this result will be divided by 9. Thus parentheses determine the order in which expressions are computed, in a manner similar to that used in algebra.

The reader should write a program similar to Example 1.4 which converts Fahrenheit to Celsius. Run it on the computer to verify that it works as expected.

b) Suppose it is required to calculate the 6th power of X, and assign this value to Y. A BASIC statement could be written as follows:

```
Y = X*X*X*X*X*X
```

However, there is a shorter way of accomplishing the same result by using the *exponentiation* operator as follows:

```
Y = X ↑ 6
```

The ↑ is a signal to the computer to perform a computation which is equivalent to multiplying the current value of X by itself 6 times.

c) *Built-In functions* can be used to perform common computations such as the square root. The statement

```
Y = SQR(2)
```

will cause the computer to use a previously programmed routine or built-in function which is designed to calculate the square root; this routine is invoked

with the use of the letters SQR. The value of the expression contained in the parentheses following SQR is called the *argument* and is the quantity which is submitted to the built-in function called SQR. In this example the argument is 2; thus Y will be assigned a value which is the square root of 2. Note that this built-in function will only calculate the square root if the argument is non-negative.

Examples of other built-in functions are ABS and LOG, for the calculation of absolute value and logarithm respectively. There are many more which will be introduced during the course of this text, with a complete list being available in the Appendices.

```
10 REM EXAMPLE 3.3
15 :
20 X = 1
25 :
30 LOOP
40   Y = X ↑ 3
50   Z = SQR(X)
60   PRINT X,Y,Z
70   X =X + 1
80   IF X > 8 THEN QUIT
90 ENDLOOP
95 :
99 STOP
```

Example 3.3 is a program which calculates the third powers and the square roots of all integers from 1 to 8 inclusive. Enter this program into the computer, run it, and observe the following output:

```
1        1        1
2        8        1.41421356
3        27       1.73205081
4        64       2
5        125      2.23606798
6        216      2.44948974
7        343      2.64575131
8        512      2.82842713
```

Note that the third powers are all printed accurately, but in most cases the square roots are terminated after 9 significant digits. Owing to hardware limitations, the computer saves only 9 digits internally, and this will be the maximum accuracy normally obtained.

3.3 Input During Execution

It is often desirable to enter data from the keyboard under program control during execution. Example 3.4 is another version of the Celsius-to-Fahrenheit conversion program which incorporates this new feature. The reader should type this program into the computer and run it to observe its operating characteristics.

```
10 REM EXAMPLE 3.4
15 :
20 LOOP
30   INPUT C
40   F = (C*9)/5 + 32
50   PRINT C,F
60 ENDLOOP
65 :
70 STOP
```

When the computer encounters the statement

```
INPUT C
```

a "?" appears on the screen. This "?" is referred to as a *prompt* and is a signal to the user to enter data. The program then pauses until the user types a number such as 110 and depresses the RETURN key. The variable C is assigned the value which has been typed, and the computer goes on to the next statement following the INPUT statement.

This program is designed to permit the user to enter Celsius data endlessly and have it converted to Fahrenheit. If you have a VIC-20, the program can be interrupted by simultaneously pressing the *STOP* key and the *RESTORE* key in response to "?". On the PET or 8032, simply press the *RETURN* key in response to "?".

Another version of the program which incorporates a *prompt message* is illustrated in Example 3.5.

```
10 REM EXAMPLE 3.5
15 :
20 LOOP
30   INPUT "ENTER DEGREES C";C
40   F = (C*9)/5 + 32
50   PRINT C,F
60 ENDLOOP
65 :
70 STOP
```

Type this example into the computer and note how much more pleasant it is to use the keyboard; you have the feeling of carrying on a "conversation" with the computer. The terms *interactive computing* or *conversational computing* are often used to refer to this type of programming technique.

Note that the INPUT statement includes the prompting message contained in quotation marks, followed by a semicolon. When executed, the prompting message is displayed followed by a question mark. The computer then expects the user to type a value for C and depress RETURN.

3.4 Selection Using IF-ELSE-ENDIF

Example 3.6 is a program which converts either Celsius to Fahrenheit or vice versa.

```
10 REM EXAMPLE 3.6
15 :
20 PRINT "TYPE 1 FOR C TO F"
25 PRINT "  OR 2 FOR F TO C"
35 LOOP
40   INPUT "ENTER 1 OR 2";T
45   IF T = 1
50     INPUT "C";C
55     F = (C*9)/5 + 32
60     PRINT C,F
65   ELSE
70     INPUT "F";F
75     C = ((F-32)*5)/9
80     PRINT F,C
85   ENDIF
90 ENDLOOP
99 STOP
```

The computer first asks the user to enter a 1 or a 2 depending on which type of conversion is to be done. This constant is assigned to the variable T. In the loop one of two separate calculations is made, depending on the current status of T. The value of T is tested using an IF statement with a relational expression as follows:

```
IF T = 1
```

If the relational expression has the value "true", the statements immediately following the IF are executed, stopping when the ELSE is encountered. If the relational expression has the value "false" the statements immediately following the ELSE are executed, up to the ENDIF. Thus the IF statement allows the programmer to logically select one of two sets of statements, depending on the value of a relational expression. When the IF-ELSE-ENDIF combination has been executed, the computer continues with the statement following the ENDIF.

IMPORTANT NOTE:

> Two distinctly separate kinds of IF statements have been introduced. The IF-THEN-QUIT is used only to terminate processing in loops and is a single statement. The IF-ELSE-ENDIF are three statements used to select two separate blocks of statements, depending upon whether the relational expression is true or false.

Enter the program into the computer and run it. Once again you can terminate its operation by depressing the RETURN key (or STOP and RESTORE) in response to ?.

3.5 Summary

The reader should now have a sufficient introduction to the fundamentals of programming in BASIC to experiment with a number of interesting problems. The following exercises which reinforce these fundamental concepts should be completed before proceeding to subsequent chapters.

3.6 Exercises

3.1 Make the following modifications to the program in Example 3.1. Enter the resulting programs into the system and run them.

a) The initial value of C is assigned in line 20. Incorporate appropriate statements so that this initial value is entered from the keyboard at execution time. A prompting message should ask the operator to enter the starting value.

b) The program as modified in a) will produce one table of output. Incorporate further statements so that the program does not stop after producing a table, but "loops", each time requesting a new starting value for C. Thus several tables can be produced. The program can be terminated by depressing RETURN (or STOP and RESTORE) when the program prompts for the value.

3.2 In Example 3.1 the conversion table is printed in increments of 10 degrees Celsius. Modify the program so that this increment changes to 2 degrees when C has reached a value of 170 degrees.

3.3 Further modify the solution for problem 3.2 so that the increment of 2 degrees is reset to an increment of 10 degrees when C has reached 180 degrees.

3.4 Write a program which permits the user to type several numbers into the computer, and receive their average as output. Note that in order to terminate the input a special number will have to be used and recognized by the program.

3.5 Suppose $100 is placed in a bank account at the beginning of each year. Interest of 8 percent is added to the account at the end of the last day of each year. There are no withdrawals. Write a program which shows the current balance in the account at the end of the first day of each year, for the first 10 years.

3.6 Modify the program for exercise 3.5 so the computer prompts the user for the interest rate, annual deposit amount, and number of years, thus making a more general-purpose program.

3.7 Marks for a particular course are assigned integer values ranging from 0 to 100 inclusive.

A mark below 50 is a failure
A mark from 50 to 59 inclusive is a D grade
A mark from 60 to 69 inclusive is a C grade
A mark from 70 to 79 inclusive is a B grade
A mark from 80 to 100 inclusive is an A grade.

Write a program which permits the teacher to enter a mark for each student in the class, and receive as output a statistical report which indicates the number of students in each of the five categories. Use appropriate prompts for the input, and headings in the output.

3.8 Modify Example 3.6 so that it terminates if a third value other than 1 or 2 is given.

3.9 Write a program which converts miles to kilometers or kilometers to miles. It should ask the user to specify what conversion should be done by specifying either a 1 or 2, and then give the value to be converted. The reader should refer to Example 3.6 for guidance. The answer should be displayed on the screen.

3.10 Modify the program in problem 3.9 so that it terminates if a third value other than 1 or 2 is given.

Chapter 4

Debugging Programs

Writing BASIC programs would be fairly straight-forward if they always ran properly. Unfortunately this is seldom the case. The process of finding errors in a program is known as *debugging*, that is, eliminating the "bugs". This process includes many different techniques for testing programs and locating errors. Some of these techniques are discussed in this chapter.

4.1 Immediate Mode

Whenever you enter a BASIC statement without a line number it is executed immediately. For example, if you type

```
PRINT "HELLO"
```

the word HELLO will be echoed back on the screen as soon as the statement is entered. This *immediate mode* of operation seems at first glance to be of limited use, since it appears that you can only execute one BASIC statement. However it turns out to be quite powerful in the preparation and debugging of programs.

4.2 An Example

```
10 REM EXAMPLE 4.1
15 :
20 I = 1
30 S = 0
35 :
40 LOOP
50   S = S + I
60   I = I + 2
70   IF I = 100 THEN QUIT
80 ENDLOOP
85 :
90 PRINT S
95 :
97 STOP
```

Suppose you have written the program in Example 4.1. You intend it to calculate the sum of all the odd integers up to 100, namely 1+3+5+7+...+99. When you place the program into execution, nothing ever appears on the screen, regardless of how long you wait. You are in an infinite loop. You can recover by depressing the STOP key, which returns you to READY status. One method of finding the bug is to stare at the program until it occurs to you what went wrong. A better way is to *examine* the current contents of the key variables to see if this gives you a clue. For example, you could type

```
PRINT I
```

Immediate-mode execution takes place and the value of I is displayed on the screen. You are shocked to find it is 493 or some other ridiculous value! How could this be, since you carefully arranged to stop the loop when the value of I reached 100? Hopefully at this point it occurs to you that I never exactly reaches 100, as it skips from 99 to 101. However, should you fail to make this observation you should insert a new line in the program, namely

```
55 PRINT I
```

and place the program into execution again. This will cause the values of I to be displayed each time through the loop. Surely now you will observe that it skips from 99 to 101, and you will then change the program by re-entering line 70 as follows


```
70   IF I > 100 THEN QUIT
```

This technique of interrupting the program and examining the values assumed by various variables is of inestimable value when debugging larger and more complex programs.

4.3 Monitoring the Operation of a Program

Some programs are expected to operate for several hours before they produce the required output. For example, you may want the computer to sort the names of all the students at the University of Waterloo into alphabetical order, and then produce a list. While the program is operating, you are worried because nothing has appeared for 30 minutes or more. You can interrupt the program with the STOP key. Then a message appears indicating the statement number which was just about to be executed, let us say number 260 to be specific. You can now "browse" around looking at partial results to see if things seem to be going well, all the time using immediate mode BASIC operations. If you are satisfied, you can resume execution at precisely the proper statement by typing

```
CONT
```

which is an immediate command to transfer control back to statement 260 and the program resumes normal operation.

4.4 Testing the Program in Small Parts

Many programs are quite large and you may wish to test only small parts of them, to be sure those parts seem to work well. Suppose in Example 4.1 you wish to test the instructions contained within the loop. You could insert a statement as follows

```
65 STOP
```

Then you could set any value you like for T and I, using immediate mode assignment statements. You could then type

```
GO TO 50
```

This would cause statements 50, 60 and 65 to be executed, and the computer would stop. Then you can examine the current value for T and I to be sure they have increased as you expected.

4.5 Summary

This chapter has only scratched the surface of methodology relative to debugging and the use of immediate-mode computing. Hopefully it contains enough guidance to enable the user to begin to develop a debugging style which will produce effective results in the shortest possible time.

4.6 Exercises

4.1 The following program prints a conversion table relating Celsius and Fahrenheit temperatures. It is supposed to print the conversion table from -20 to 100 degrees Celsius in increments of 10 Celsius degrees except for the comfort range from 10 to 35 degrees where it should print the temperatures for every 2 degrees Celsius. After 35 degrees the table is again printed in 10 degree increments. Run this program and examine the output. Does it work? Attempt to repair the program so it will work. If you have difficulties, use the debugging techniques described in this chapter to help you.

```
C = -20
D = 10
PRINT "C", "F"
LOOP
  IF C=100 THEN QUIT
  F = (C*9)/5 + 32
  PRINT C,F
  IF C = 10
    D = 2
  ELSE
    IF C = 35
        D = 10
    ENDIF
  ENDIF
  C = C + D
ENDLOOP
STOP
```


4.2 The program similar to the one in Exercise 4.1 prints a conversion table relating Celsius and Fahrenheit temperatures. It is supposed to print the conversion table from -20 to 100 degrees Celsius in increments of 10 Celsius degrees except for the range from 0 to 5 degrees where it should print the temperatures for every .2 Celsius degrees. After 5 degrees the table is again printed in 10 degree increments. Run the program and examine the output. Does it work? Find the bugs and fix them. If you have difficulties use the debugging techniques described in this chapter.

```
C = -20
D = 10
PRINT "C", "F"

LOOP
  IF C = 100 THEN QUIT
  F = C(C*9)/5 + 32
  PRINT C,F
  IF C = 0
    D = .2
  ELSE
    IF C = 5
      D = 10
    ENDIF
  ENDIF
  C = C + D
ENDLOOP
```

Chapter 5

Arithmetic in BASIC

In earlier chapters there have been several examples of arithmetic operations within BASIC programs. The purpose of this chapter is to approach the subject more formally, in order to tidy up many of the ideas which have been introduced.

5.1 Numeric Constants

BASIC programs and the data they process usually contain many *numeric constants*, several examples of which follow:

(a)
- (i) 126
- (ii) -126
- (iii) 0
- (iv) 111222333
- (v) -111222333

(b)
- (i) .126
- (ii) -.126
- (iii) .0
- (iv) 111222.333
- (v) -111222.333

(c)
- (i) 12.6 E3
- (ii) 12.6 E-3
- (iii) -12.6 E3
- (iv) -12.6 E-3
- (v) 999.999999 E13
- (vi) 1.0E33
- (vii) 1.0E-34

Group a) contains a number of examples of *integers*. Integers contain no decimal point, as it is assumed to be to the immediate right of the least significant digit. They can be positive or negative, and can contain at most 9 digits as shown in examples a) (iv) and a) (v).

Group b) contains several examples of *real* constants. Each contains a decimal point, can be positive or negative, and has at most 9 digits of precision.

Group c) contains several examples of real constants expressed using *exponent* or *scientific* notation, usually referred to as E-notation. In all cases the signed integer following the E indicates the power of ten which is used to multiply the number which precedes it. For example 12.6E3 means 12.6 multiplied by 10 to the third power and is therefore equivalent to 12600. On the other hand 12.6E-3 is equivalent to .0126 because the -3 indicates that 12.6 should be multiplied by 10 to the -3rd power, which is equivalent to dividing by 10 to the 3rd power.

E-notation is used to allow the programmer to use very large or very small constants in the program or data. Example c) (vi) is a number which is so large it could almost be considered infinite. On the other hand example c) (vii) shows a number which is so small it could almost be considered to be zero. All E-notation constants can contain up to 9 digits of precision. The largest magnitude is approximately 10 to the 38rd power, and the smallest magnitude, other than zero, is approximately 10 to the -38th power.

The following are examples of errors which can be made when using numeric constants in BASIC.

142.6.7	Two decimal points
1,462,271.439	Cannot use commas
$126.42	Cannot use $
14.E16.2	Cannot use decimal in exponent part
14.6E93	Too large (overflow)

5.2 Numeric Variables

Numeric variables are used in BASIC for the purpose of being assigned numeric values. Consider the following examples.

a) In the statement

```
X = 7.6
```

the symbol X is a numeric variable, and is assigned the value 7.6 when the statement is executed.

b) In the statement

```
INPUT Y
```

the symbol Y is a numeric variable, and is assigned a value when any numeric constant is entered at the keyboard during execution.

c) In the statement

```
PRINT K
```

the numeric variable K has previously been assigned a numeric value, and this value is displayed on the screen.

A numeric variable always begins with a letter, and can contain as many letters or digits as can conveniently be used on the line being entered into the computer (in other words, a numeric variable cannot by typed partly on one line, and partly on the next). Examples of numeric variables are as follows:

(i) CAT
(ii) G72
(iii) THISISABIGVARIABLE

Notes:

(1) Only the first two characters in a real variable name are used by the computer. Consequently if you use the variable names CAT and CA in your program the computer assumes they are the same variable. The reason more than two characters are allowed is to permit meaningful variable names to be used. Obviously this must be done with great care if errors are to be avoided.

(2) Variable names cannot be one of the reserved words in BASIC. Thus SAVE is not a permissible variable name, nor is LOAD, PRINT or

STOP. If you inadvertently use one of these names the computer will display the message

SYNTAX ERROR IN LINE XXX.

(3) Variable names cannot *contain* any of the BASIC reserved words. Thus the variable LOADER is improper since it contains LOAD. Also RERUN is improper because it contains RUN. In these cases the computer will also display the message

SYNTAX ERROR IN LINE XXX.

Examples of other erroneous numeric variable names are as follows:

7K Must begin with a letter
C$K Must not contain the special character $
WORD Contains the reserved word OR.

5.3 Numeric Expressions

Any expression which, when evaluated, produces a numeric result is said to be a *numeric expression.* Some examples are:

(i) (X + 6.92)/8.6
(ii) (A + T)*(J - 2)
(iii) 7.493
(iv) K
(v) -K
(vi) A+B↑5
(vii) A+B↑.5
(viii) A/B*C
(ix) -7.9↑2

The conventions used in evaluating numeric expressions are similar to those used in algebra. For example in (i) the expression X + 6.92 would be evaluated first because it is contained in parentheses. The result would then be divided by 8.6 (recall that the slanted stroke "/" is used to denote the division operator).

Example (ii) introduces "*" as the multiplication operator. Here the two expressions A + T and J - 2 are evaluated, and the two results are multiplied together. Note that in BASIC the multiplication operator "*" must

always be used when multiplication is to be done, whereas in algebra we are allowed an implied multiplication, with the expression written as (A+T)(J-2).

Examples (iii) and (iv) are included to show that the simple numeric constant and numeric variable are each simple cases of numeric expressions.

Example (v) introduces the *unary minus.* This operator is the same symbol as the subtraction operator, but serves to multiply the value of K by negative or minus one, thus changing its sign.

Example (vi) introduces the exponentiation operator "↑". First B↑5 is evaluated to produce B multiplied by itself five times. Then the result is added to A. This is because of the *priority* of operators. The operators in order of decreasing priority are

↑
* and /
+ and -

Example (vii) indicates the use of a fractional exponent, namely .5. Here B is raised to the power .5 to produce the square root of B; the result is added to A. Note that B must be positive or an error message would be produced by the computer.

Example (viii) seems to provide two possible results. Will A/B be evaluated, with the result multiplied by C? Or will B*C be evaluated with the result divided into A. The rule is that when equal priority operators are encountered the expression is evaluated from left to right, so the A/B is evaluated then multiplied by C. It is usually best to use parentheses and then no possible misunderstanding can occur. Thus the expression could be written as (A/B)*C.

Example (ix) produces the negative of 7.9 squared. The expression (-7.9)↑2 would produce the same result with a positive sign.

5.4 Numeric Assignment Statements

The general form of a *numeric assignment statement* is

numeric variable = numeric expression.

First, the value of the numeric expression is computed; this value is then assigned to the numeric variable.

Examples are:

```
(i)   SALARY = HOURS * 6.25 - DEDUCTS
(ii)  AGE = 7
```

An alternative form of the assignment statement is to use the BASIC keyword LET as follows:

```
(i)   LET SALARY = HOURS * 6.25 - DEDUCTS
(ii)  LET AGE = 7
```

An error frequently caused by beginning programmers is to try to use an assignment statement such as the following:

```
X + Y = 7 * Z
```

The item to the left of the equals sign is not a variable. You cannot have an expression to the left of the equals sign when you are using an assignment statement.

5.5 Some Examples

a) Suppose a loan of $10,000 is to be repaid at the rate of $750.00 per month, payable at the end of each month. This payment is to include interest of 1 percent per month on the outstanding portion of the loan (balance). Example 5.1 is a program which calculates the number of months required to repay the loan. It also prints out a schedule to show the declining balance, and interest payments.

```
100 REM EXAMPLE 5.1
110 :
130 BAL = 10000
140 M = 1
160 LOOP
170   I = BAL * .01
190   PRINT M;BAL,I
200   BAL = BAL - (750 - I)
210   M = M + 1
220   IF(BAL*(1+.01))<=750 THEN QUIT
230 ENDLOOP
240 I = BAL * .01
260 PRINT M;BAL,I
270 PRINT
280 PRINT "MONTHS TO REPAY IS";M
300 STOP
```

The first few lines in the program which appear before the main loop are used for *initialization*. The numeric variables BAL and M are assigned their initial values of 10000 and 1 respectively.

Each time the LOOP is executed the payment schedule for one month is computed and printed. Then the new balance is recalculated and the month is incremented by one. The loop is terminated by testing to determine whether the current balance plus one month's interest (BAL * (1+.01)) can be covered by the payment of $750. If so, the final line is calculated and printed after the exit from the loop. Finally a message is printed which indicates the total number of months in which payments must be made to repay the loan completely. The output produced by the computer appears as follows:

```
 1   10000          100
 2   9350  93.5
 3   8693.5         86.935
 4   8030.435       80.30435
 5   7360.73935     73.6073935
 6   6684.34674     66.8434674
 7   6001.19021     60.0119021
 8   5311.20211     53.1120211
 9   4614.31413     46.1431413
10    3910.45728    39.1045728
11    3199.56185    31.9956185
12    2481.55746    24.8155746
13    1756.37304    17.5637304
14    1023.93677    10.2393677
15    284.176139    2.84176139

THE NUMBER OF MONTHS TO REPAY IS 15
```

Notes:

(1) Quantities may be recorded to several decimal places, even though they represent dollars and cents. In a subsequent section it is shown how to truncate and round these numbers.

(2) The reader should note that the decimal points do not line up vertically. This is because all numeric output is left-justified in the print window.

(3) In line 270 the PRINT keyword has no associated list of variables or constants. When the computer executes this statement a line of blank spaces is printed.

b) Example 5.2 is identical to the previous example, except that the program contains an important change of *style*. Most of the numeric constants have been replaced by numeric variables which are initialized *once* in the initialization portion of the program. For example, each time .01 appears it is replaced by RATE. This makes the program much easier to modify when these important parameters change. For example, if the interest rate becomes 2 percent, it is only necessary to change one line in the initialization, namely

```
RATE = .02
```


The program will now function correctly and produce a corresponding result for the new rate.

This point of *programming style* cannot be emphasized too much. Often programs are used repeatedly for years by different people. These people or *users* wish to introduce different data (such as interest rates), so the initial assignments of variables will have to be changed. The effort involved to make this change becomes relatively trivial if all such changes can be isolated in the initialization portion of the program.

```
100 REM EXAMPLE 5.2
105 :
110 PAY = 750
120 RATE = .01
130 BAL = 10000
140 M = 1
160 LOOP
170   I = BAL * RATE
190   PRINT M;BAL,I
200   BAL = BAL - (PAY - I)
210   M = M + 1
220   IF(BAL*(1+RATE))<=PAY THEN QUIT
230 ENDLOOP
240 I = BAL * RATE
260 PRINT M;BAL,I
270 PRINT
280 PRINT "MONTHS TO REPAY IS";M
300 STOP
```

5.6 Numeric Built-In Functions

In Chapter 3 the built-in function SQR was introduced, and others such as ABS and LOG were mentioned. A complete list of available functions and their characteristics can be found in the Appendices. However, a number of observations are useful.

Every function such as SQR can be considered to be a numeric variable which is assigned its value when the function is encountered. Thus it is not possible to use the numeric variable name SQR for any other purpose, as it is "reserved" for use as the built-in function. The variable name SQR can only be used in conjunction with parentheses which contain the

proper *argument*. (Note that a few of the functions used in later chapters can contain more than one argument.)

The arguments used in functions can be expressions. For example SQR(X + Y * Z) is evaluated by first computing the expression X + Y * Z, followed by the evaluation of the square root. Note that the value of the expression must be non-negative in order for the SQR function to produce a result.

The arguments can also be expressions which contain another function. For example SQR(SQR(2)) will produce the 4th root of 2.

5.7 Integer Computations in BASIC

Sometimes it is important to do arithmetic which ignores the fractional component of a numeric quantity. Consider the following example. Suppose you wish to calculate the maximum number of quarters (25-cent pieces) contained in the quantity $3.15. Obviously the answer is 12 with $.15 left over. To compute this the following BASIC statements could be incorporated into a program.

```
QUARTERS = 315/25
PRINT QUARTERS
```

Unfortunately the answer would appear on the screen as 12.6 because it is indeed true that $3.15 contains 12.6 quarters! However, since .6 of a quarter is not legal currency, 12 is the required answer. To obtain an integer answer for the number of quarters the following statements are used:

```
QUARTERS = INT(315/25)
PRINT QUARTERS
```

Note that the built-in function INT (which stands for "Integer Part") has been used. First the expression 315/25 is computed to produce 12.6, and then the *integer part*, namely 12 is selected and is finally assigned to the variable QUARTERS.

Many people have a habit of collecting one-cent coins in an old cigar box, usually located in the bedroom. This habit drives the government to distraction, but prevents many holes from developing in pants' pockets. Example 5.3 is a program which permits the user to indicate the number of cents in the collection. The program proceeds to compute the minimum

number of coins required to make up this sum, using 5 denominations, namely 50-cent pieces (halves), 25-cent pieces (quarters), 10-cent pieces (dimes), 5-cent pieces (nickels) and 1-cent pieces (pennies). The reader should study the program to observe the use of the INT built-in function.

```
100 REM EXAMPLE 5.3
110 PRINT "HOW MANY CENTS?"
120 INPUT AMT
130 H = INT(AMT/50)
140 BAL = AMT - 50*H
150 Q = INT(BAL/25)
160 BAL = BAL - 25*Q
170 D = INT(BAL/10)
180 BAL = BAL - 10*D
190 N = INT(BAL/5)
200 P = BAL - 5*N
205 PRINT
210 PRINT AMT;"CENTS COULD BE:"
220 PRINT H,"HALVES"
230 PRINT Q,"QUARTERS"
240 PRINT D,"DIMES"
250 PRINT N,"NICKELS"
260 PRINT P,"PENNIES"
270 STOP
```

When the program is run the following output is produced when the number of cents in the collection is 280.

```
HOW MANY CENTS?
? 280

280 CENTS COULD BE:
 5          HALVES
 1          QUARTERS
 0          DIMES
 1          NICKELS
 0          PENNIES
```

5.8 Rounding, Truncating and User-Defined Functions

In example 5.1 and 5.2 the output contains several decimal places, even though it involves money which properly should be printed to the nearest cent. Consider the following sequence of BASIC statements

```
10 X = 12.3456789
20 X = X + .005
30 X = X * 100
40 X = INT(X)
50 X = X/100
60 PRINT, X
```

(i) Statement 20 causes rounding to take place to the nearest cent.

(ii) Statement 30 causes all dollars and cents to appear in the integer part of X because of the multiplication by 100.

(iii) Statement 40 causes the fractional part of X to be omitted.

(iv) Statement 50 shifts the decimal to the left by two places, thus causing the "cents" to be to the right of the decimal.

This series of statements will cause X to be rounded to the nearest cent before printing. When line 60 is executed the value 12.35 will be printed.

The four "rounding" statements can be combined into one as follows:

```
X = INT((X + .005)*100)/100
```

Since this is a very common requirement, it is fortunate that BASIC permits us to define a *user-defined function* with a single statement as follows:

```
DEF FNA(X) = INT((X + .005)*100)/100
```

This DEF statement defines a function FNA which has a parameter X.

Consider Example 5.4 which incorporates this function definition in line 110. The function is used in lines 190 and 260 in the same manner as built-in function SQR. The variables BAL and I are used as arguments in separate invocations of FNA, and their rounded, two-decimal version is returned as the value of FNA.

All user-defined function names begin with FN. What follows (in this case the letter A) is a properly constituted variable name.

Note that user-defined functions are a single BASIC statement which has the following format:

DEF function name = expression

```
100 REM EXAMPLE 5.4
105 :
110 DEF FNA(X) = INT((X+.005)*100)/100
115 :
130 BAL = 2000
140 M = 1
150 PRINT "MONTH","BALANCE","INTEREST"
155 :
160 LOOP
170   I = BAL * .01
190   PRINT M,FNA(BAL),FNA(I)
200   BAL = BAL - (200 - I)
210   M = M + 1
220   IF (BAL*(1+.01))<=200 THEN QUIT
230 ENDLOOP
235 :
240 I = BAL * .01
245 :
260 PRINT M,FNA(BAL),FNA(I)
270 PRINT
280 PRINT "MONTHS TO REPAY IS";M
285 :
300 STOP
```

5.9 Summary

Arithmetic operations appear in arithmetic expressions which include operands, operators, built-in functions and parentheses. The operands are either numeric constants or numeric variables. The operators are +, -, *, /, and ↑. All calculations proceed retaining 9 digits of precision.

Results of arithmetic operations can be assigned to a numeric variable. However they can be printed directly without assignment by including them in the list of items specified in a PRINT statement. For example, the following statement would produce the result of the expression X+2.

```
PRINT X+2
```

5.10 Exercises

5.1 A retired school teacher has $60,128.42 in his bank account. Interest of one-half of one percent is credited at the end of each month. Assume the teacher must withdraw $1,020.00 for living expenses at the beginning of each month. Write a program which calculates the number of months until the bank account has reached a balance which is less than $1,020.00.

5.2 Write a program which converts a time in seconds to a time in hours, minutes and seconds. The program should continue in a loop; each time the user is requested to type a time in seconds, and the converted time is printed on the screen.

5.3 Write a program which computes and prints all the integers up to 1000 which are perfect squares. (625 is a perfect square because its square root is 25, which is an integer.)

5.4 Write a program which converts a distance in inches to a distance in miles, yards, feet and inches. The program should continue in a loop; each time the user is requested to type a distance in inches, and the converted distance is printed on the screen.

Chapter 6

Hardware-Dependent Limitations

When using a computer, sometimes surprising and unexpected results can occur. These are usually related to the limitations of the hardware, and it is helpful to be forewarned. This chapter outlines some of these problems, and gives partial explanations. It is possible to skip this chapter on an initial reading of the text, as future chapters do not depend on this material.

6.1 An Example

```
10 REM EXAMPLE 6.1
15 :
20 X = 2.0
25 :
30 LOOP
40   Y = SQR(X)
50   PRINT X,Y
60   X = X + .1
70   IF X = 3.0 THEN QUIT
80 ENDLOOP
85 :
90 STOP
```

Consider Example 6.1 which tabulates the square roots of the set of numbers 2.0, 2.1, 2.2, 2.3, ..., 3.0. It would be reasonable to expect this program to terminate but in fact it does not.

This problem is caused because the computer represents numbers using binary notation and it is not possible to represent .1 accurately using this notation. Since the computer must use an appproximation for

.1 it follows that X never actually becomes equal to 3.0 and the program does not terminate. The program is easily corrected by changing line 70 as follows:

```
70   IF X > 3.0 THEN QUIT
```

It is therefore common to avoid equality tests when terminating loops. However, if the numbers involved are all integers this problem does not arise (unless the integers are very large). This is because most integer values are represented accurately in binary.

To further help the reader understand the problem, perhaps an analogy using decimal arithmetic would be of assistance. Suppose it is required to write the fraction one third as a decimal. It is written as .3333333, a number which never terminates; thus no computer could ever have enough capacity to accurately represent this fraction, if the computer recorded its numbers in common decimal notation. If we accept .3333333 as the approximation it is slightly too small, but the error is less than 1 part in ten million.

6.2 Another Example

```
10 REM EXAMPLE 6.2
15 :
20 X = 3.1
30 Y = 3.1E20
40 Z = Y - Y + X
50 T = Y + X - Y
55 :
60 PRINT Z,T
65 :
70 STOP
```

In Example 6.2 you would expect Z and T to be assigned identical values. In actual fact, Z becomes 3.1 and T becomes zero! This happens because computation of expressions proceeds from left to right, and numeric constants contain only 9 digits of precision. When computing Z, the quantity Y-Y is evaluated first, yielding zero; then X is added to produce the result 3.1. When computing T, the quantity Y+X is evaluated first. Since the computer retains only 9 significant digits, the result is 3.1E20 because X is insignificant relative to Y. Then Y is subtracted producing the zero

result for T. This points out that operations which are associative in ordinary algebra are not necessarily associative in BASIC.

6.3 Summary

The reader who is not familiar with computers may be disturbed by the points made in the two examples. However, experienced computer users have learned over the years to cope with these difficulties, and they seldom present problems in straight-forward real-life situations. The most serious difficulties arise in complex scientific or engineering calculations where millions of computations are taking place. Since the numbers used are approximations, the errors can have a tendency to compound upon one another. In extreme cases the error becomes larger than the numbers themselves, so the results are meaningless! A separate discipline called numerical analysis examines the propagation of errors. These studies have yielded good algorithms for solving common scientific problems keeping the error in the results to a minimum.

Chapter 7

String Manipulation in BASIC

Many problems involve the processing of alphabetic data. Examples include names, addresses, and product descriptions, to name just a few. BASIC permits the programmer to manipulate alphabetic data with reasonable ease. The purpose of this chapter is to introduce the subject of *string processing*, and to formalize a number of the pertinent rules of BASIC.

7.1 String Constants

In previous chapters there have been examples in which headings were produced by printing strings of characters contained between quotation marks. Examples of these *string constants* are as follows:

```
(i)     "DOGS"
(ii)    "CAT"
(iii)   "   "
(iv)    ""
(v)     "IT'S A BOY!"
```

String (i) and (ii) have *lengths* of 4 and 3 respectively.

Example (iii) shows a string of 3 blank characters, while example (iv) illustrates the *null string*. This null string is of length zero and contains no characters.

Example (v) illustrates what to do if a string is to contain quotation marks as in the word IT'S. Double quotes are used to delimit the string, and the single quote is used within it. A string can have up to 255 characters.

7.2 String Variables

A *string variable* is a variable which can assume a string constant as its value. String variables are similar to numeric variables except that they *always* end with a "$" character. Thus the following are typical string variables:

```
A$
NAME$
CTR$
```

As with numeric variables, they must begin with a letter, and contain no special characters except of course the final $ character. Only the first 2 characters plus the $ are actually used by the computer.

7.3 String Expressions

There is only one string operator, namely *concatenation*, which is denoted by a "+". For example,

```
"CAT" + "DOG"
```

is a string expression which, when executed, causes the two strings to be joined together (concatenated) to form a single string, namely "CATDOG". In a similar fashion, three strings can be concatenated as follows:

```
"CAT" + " " + "DOG"
```

to form the string "CAT DOG". As many strings can be concatenated together as are required in a particular problem.

String variables and string constants can also appear in the same string expression. For example,

```
"CAT" + X$
```

will concatenate the string "CAT" to the current value assigned to the string variable X$, thus forming a new string.

The following are examples of invalid string expressions.

"CAT" + 6	Cannot concatenate a string with a numeric constant
"CAT" + DOG	Here, DOG is not contained within quotes so is not a string constant
"CAT" * "DOG"	The * operator is not allowed in string expressions.

Later in this chapter built-in string functions are introduced; they can also be used in string expressions.

7.4 String Assignment Statements

The general form of a *string assignment statement* is

string variable = string expression

Examples are:

```
(i)    X$ = "CAT" + "DOG"
(ii)   NAME$ = "JOHN HENRY"
(iii)  T$ = X$ + NAME$
(iv)   T$ = T$ + T$
```

In each example the string variable on the left of the equals sign is assigned the value of the string expression on the right. Note in example (iv) the variable T$ is concatenated to itself to form a string of twice the original length.

7.5 An Example

```
10 REM EXAMPLE 7.1
15 :
20 PRINT "WHAT IS YOUR SURNAME?"
30 INPUT LAST$
40 PRINT "AND YOUR FIRST NAME?"
50 INPUT FIRST$
60 PRINT "YOUR MIDDLE INITIAL?"
70 INPUT MIDDLE$
75 :
80 FULL$=FIRST$+" "+MIDDLE$+". "+LAST$
90 PRINT FULL$
95 :
99 STOP
```

Example 7.1 is a program which prompts the user for surname, first name and middle initial. The three items are entered at the keyboard, and are assigned to three separate string variables. The full name is formed by concatenating the three strings together in the proper order. Note that a period is placed after the initial, and blank characters are inserted between the components of the full name.

When the program is run, the user is prompted to enter three string constants. To make things easier, it is not necessary to use the beginning and ending quotes around the input string. Thus when the surname is requested you can type either "SMITH" or SMITH as a response. (When you choose to omit the quotation marks, the string constant must *not* contain a comma (,) as this becomes a delimiter. This situation is discussed further in Chapter 8.)

The computer *never* prints strings on the screen (or on any file device) with the quotes included. Thus when the full name is printed it appears in a normal format, for example,

```
JOHN H. SMITH
```

It should be noted that in Chapter 2 we discussed the concept of print "windows" which are 10 characters wide. If a string of 10 characters or longer is printed, it uses two or more of these windows. In fact, if a string is longer than a line, the output is automatically continued on the next line, and is said to "wrap around".

7.6 Built-In Functions for Strings

While built-in functions play a minor role in most numeric calculations, it is rare that a string manipulation operation can be done effectively without the use of built-in functions. Hence these functions play an important role, and appear in many string expressions. The following examples illustrate the application of some of the more common functions. Others can be found in the Appendices.

a) It is often necessary to select a string of characters from within an existing string, thus forming a new string, often referred to as a *substring*. This is accomplished using the MID$ function with three arguments as follows:

```
Y$ = "ABCDEFGH"
X$ = MID$(Y$,3,4)
```

The string Y$ is selected, and the new string is formed beginning at the 3rd character, namely the C, and continuing for 4 characters. Thus the string X$ will be assigned the value "CDEF" and will have a length of 4.

```
10 REM EXAMPLE 7.2
15 :
20 PRINT "ENTER A 3-LETTER WORD"
30 INPUT X$
35 :
40 L1$=MID$(X$,1,1)
50 L2$=MID$(X$,2,1)
60 L3$=MID$(X$,3,1)
70 STRING$=L1$+" "+L2$+" "+L3$
75 :
80 PRINT STRING$
85 :
90 STOP
```

Example 7.2 illustrates a simple application of this function. It requests the user to enter a three-letter word such as CAT. The program then composes a new string, placing a blank between each letter, and the screen output becomes C A T.

b) The previous example is somewhat restrictive in application because it processes only 3-letter words. In order to be able to input a word of any

length, it would be convenient to have a facility which determines the length of any given string. The built-in function LEN is used for this purpose. For example

```
INPUT X$
N =LEN(X$)
Y$=MID$(X$,N,1)
PRINT Y$
```

will permit the user to enter a string constant of any reasonable length; it will be assigned to X$, and then the length is determined and is assigned to N. Finally the *last* character in the string is selected using the MID$ function, and it is printed on the screen.

```
10 REM EXAMPLE 7.3
15 :
20 PRINT "PLEASE ENTER A WORD"
30 INPUT X$
35 :
40 N=LEN(X$)
50 Y$=MID$(X$,1,1)
60 I=2
70 LOOP
80   IF I>N THEN QUIT
90   Y$=Y$+" "+MID$(X$,I,1)
91   I=I+1
92 ENDLOOP
93 :
94 PRINT Y$
99 STOP
```

Example 7.3 uses the LEN function to generalize Example 7.2 so that a string of any reasonable length can be entered. The output is the same string, with a single blank inserted between each character. Study this example carefully to be sure the algorithm is thoroughly understood, because the technique illustrated is commonly used in string processing.

Initially Y$ is a string of length 1 which contains the first character in X$. Each time the loop is executed a blank character and the next available character in X$ is concatenated on the right. When all characters in X$ have been used the loop terminates and the resulting string Y$ is printed. Note that the input word must contain at least one character.

c) Example 7.4 is another application of the MID$ function. Here the user is asked to input a sentence and the program is to determine the number of occurrences of the letter A. This is done by comparing every individual character in the sentence with "A", and adding unity to N for each equal comparison.

```
100 REM EXAMPLE 7.4
110 PRINT "ENTER A SENTENCE"
120 INPUT S$
130 N=0
140 L=LEN(S$)
150 I=1
160 LOOP
170   IF MID$(S$,I,1)="A"
180     N=N+1
190   ENDIF
200   I=I+1
210   IF I>L THEN QUIT
220 ENDLOOP
230 PRINT "A OCCURS ",N," TIMES"
240 STOP
```

d) Sometimes the programmer would like to include a numeric constant as part of a character string. Consider the example:

```
N = 256
Y$ = "TRY " + N " TIMES"
PRINT Y$
```

Here the desire is to print the message

```
TRY  256 TIMES
```

However an error occurs because the string expression contains both string constants and a numeric variable N. The numeric expression N can be *converted* to a string expression using the STR$ function as follows:

```
N = 256
Y$ = "TRY " + STR$(N) + " TIMES"
PRINT Y$
```

Now the desired result is achieved. A similar function VAL permits the programmer to convert a string constant containing a properly formed numeric constant so it can be processed in arithmetic expressions. Consider the example

```
X$ = "123.4"
Y = VAL(X$)*2
PRINT Y
```

Here the string constant "123.4" is converted to a numeric constant 123.4 using the VAL function. It is then multiplied by 2, and the final result is printed.

```
100 REM EXAMPLE 7.5
110 PRINT "ENTER A SENTENCE"
120 INPUT S$
130 N=0
140 L=LEN(S$)
150 I=1
160 LOOP
170   IF MID$(S$,I,1)="A"
180     N=N+1
190   ENDIF
200   I=I+1
210   IF I>L THEN QUIT
220 ENDLOOP
230 PRINT "A OCCURS "+STR$(N)+" TIMES"
240 STOP
```

Example 7.5 shows another situation in which you may want to use the STR$ function. This program is identical to that of Example 7.4, except that the PRINT statement has been altered to use the function. Note that only one string is printed, thus avoiding the "windows" problem, and providing output with a nicer format. Try the programs and observe the results in each case.

e) The LEFT$ function permits the programmer to select a specific number of characters from the left end of a string. Consider the statements:

```
10 X$ = "ABCDEFG"
20 Y$ = LEFT$(X$,3)
30 PRINT Y$
```


The value ABC would be printed because the 3 characters are selected from the left of X$. Note that LEFT$ has two arguments.

The RIGHT$ function is similar except that the required number of characters is selected on the right of the string.

7.7 Summary

Note that when a string function returns a string as its value, the function name always ends in a $, for example MID$. However, when the function returns a numeric result (such as LEN) the function name does *not* end with a $. Properly speaking, the function LEN is not a string function at all, but is a numeric function. However, it is included in this chapter because its use is associated with strings.

7.8 Exercises

7.1 a) Write a program which inputs a three character string such as CAT and prints the three letters vertically as follows:

```
C
A
T
```

b) Modify the program in a) so that the string can be of any reasonable length.

7.2 a) Write a program which inputs a collection of words one at a time from the keyboard, and determines the total number of words of various lengths. The output will be a table as follows:

```
ONE CHARACTER  = 3
TWO CHARACTERS = 2
THREE CHARACTERS = 6
MORE THAN THREE = 12
```

b) Modify the program for part a) so that several words can be entered on a single line with each word separated by one or more spaces.

7.3 With reference to the Exercise 7.2, modify the output to appear as follows:

```
NUMBER OF CHARACTERS

 1   ***
 2   **
 3   ******
>3   ************
```

The number of asterisks corresponds to the number of occurrences of a word of the size indicated.

7.4 Write a program which has the following characteristics:

a) Words are read one at a time from the key-board of the computer.

b) These words are arranged into lines for printing with no line being longer than the width of the screen. The first word in a line is printed left-justified in the line. All subsequent words in a line are separated from other words by exactly one blank character. If there is not enough space at the end of a line to print the next word, this space should be left blank, with the next word appearing as the first word in the following line.

c) When a period (.) is read from the keyboard the program terminates.

7.5 Write a program that accepts two integers as input from the keyboard. The program then must print a rectangle with the two integers as the length and width, using an asterisk (*) as the character to outline the rectangle. For example, if 5 and 8 are read the output should appear as follows:

```
* * * * * * * *
*               *
*               *
*               *
* * * * * * * *
```


7.6 Write a program which reads a message containing only letters and blank spaces. This message is to be coded into a "secret message" using the following coding rules:

(i) Any letter is replaced by its successor in the alphabet. For example, A is replaced by B and S by T. The letter Z is replaced by A.

(ii) Blanks are to remain unchanged.

Note: You will find the built-in functions ASC and CHR$ useful for solving this problem. They are described in the Appendices.

Chapter 8

Simple Input, Output and Files

In all examples introduced to this point, the BASIC program caused printing to take place on the screen in 10-character "windows". Also, data has been read into the computer using the INPUT command and the keyboard. The purpose of this chapter is to generalize these ideas, and specifically to introduce the concept of a *file.*

8.1 Output on the Screen

Several examples have been introduced where a number of quantities are printed on the screen of the computer. For example, the statement

```
PRINT X$, "CAT", Y, 6.47
```

will cause four values to be displayed in four 10-character windows. First the current value of X$ is printed in the first window, assuming it has a length less than 10. The string CAT is printed in the second window, the value of Y in the third, and the numeric constant 6.47 in the fourth. All quantities are left-justified in each window. If the length of X$ is 10 or greater, as many windows as necessary are used. If there are not enough windows on the printed line, the computer continues printing on the next and subsequent lines until all the quantities are printed.

The list of items following the keyword PRINT is referred to as the *output list.* The elements of the list are separated by commas and are referred to as *output-list items*, or *list items* for brevity.

Each list item can be a variable name, string constant, numeric constant or expression. If we execute the statement

```
PRINT (X+2)*6.3
```

the expression (X+2)*6.3 is evaluated and the result is printed.

8.2 Input using the Keyboard

The INPUT keyword has been used in many examples. When the statement

```
INPUT X
```

is executed, a "?" appears on the screen, and the user is expected to type some valid numeric constant. If the user types an invalid quantity, such as CAT, the following error message is displayed

```
?REDO FROM START
```

You must then enter a valid numeric quantity to continue.

It is possible to input a list of quantities as follows:

```
INPUT X, Y$, T
```

Here the terminal prints the "?" and the user normally is expected to type three quantities, separated by commas. A typical response might be

```
26.49, CAT, 16
```

in which case X will be assigned the value 26.49, Y$ the string value "CAT", and T the value 16. It is important to observe that the input quantities must match the input list items in both number and type. String constants do not require the quotes on input; however, if the string to be assigned to Y$ contains a comma or a space, then it must be typed surrounded by quotes. Thus the response

```
26.49, "JONES,HENRY",16
```

will assign the string value JONES,HENRY to the variable Y$.

It is not absolutely necessary to type all three items of data on one line, separated by commas. They could be placed on three separate lines, or two on one line and one on the other. However, when more than one data item appears on a line, the items on that line must be separated by commas. The computer will display a double question mark ?? until sufficient data has been entered.

8.3 Files in BASIC

It is frequently desirable to store data on an *external device* such as a cassette. For example, a *file* could be created which contains the names of all the students in a class. This file would contain several *records*, one for each student. The file is given a name such as STUDENT.

The files are stored on cassette in a form similar to that used to place music on a tape for use with a tape recorder. Magnetic impulses are written which are coded to represent the various characters.

```
0110,STEVENS        ,M,17,065,063,085,056,076
0297,WAGNER         ,M,15,065,086,085,084,074
0317,RANCOURT       ,F,16,075,072,070,068,065
0364,WAGNER         ,M,16,070,058,090,064,083
0617,HAROLD         ,M,17,085,080,080,075,074
0998,WEICKLER       ,M,16,072,074,075,075,075
1203,WILLS          ,F,16,073,072,072,073,084
1232,ROTH           ,M,17,072,070,070,074,072
1234,GEORGE         ,M,18,070,070,071,058,069
1265,MAJOR          ,M,16,065,065,068,068,069
1568,POLLOCK        ,M,17,089,088,085,092,063
1587,PEARSON        ,F,15,055,050,049,061,060
1617,REITER         ,M,17,100,068,069,075,089
2028,SCHULTZ        ,M,18,069,068,075,074,053
2036,BROOKS         ,M,18,065,068,069,070,065
2039,ELLIS          ,M,17,085,085,085,085,085
2049,BECKER         ,F,15,065,065,065,068,069
2055,ASSLEY         ,M,16,065,063,060,063,065
2087,STECKLEY       ,M,15,056,053,085,084,072
9999,ZZZZ           ,M,99,000,000,000,000,000
```

Figure 8.1

Consider Figure 8.1 which is a listing of a file called STUDENT. This file contains 20 records. Each record contains several *fields* of information about a student. These fields are student-number, surname, sex, age, and the marks obtained in 5 courses, namely Algebra, Geometry, English,

Physics, and Chemistry. Note that each field quantity terminates with a comma (except for the last one). This comma is used as a delimiter to separate the fields.

Another thing to observe is that the 20th record is a *sentinel* record which is used to define the end of the file. This record contains a special student number, namely 9999. The entire record is a "dummy" which exists only to indicate that no further records follow in the file. It will not be processed normally, but will be used as a signal to terminate processing when the file is being read.

8.4 Reading a File

Example 8.1 is a program which reads the STUDENT file stored on cassette and prints out a list of the students' names.

```
10 REM EXAMPLE 8.1
15 :
20 OPEN 3,1,0,"STUDENT"
25 :
30 LOOP
35   INPUT#3,NO,N$,S$,A,M1,M2,M3,M4,M5
40   IF NO=9999 THEN QUIT
45   PRINT N$
50 ENDLOOP
55 :
60 CLOSE 3
65 :
70 STOP
```

In statement 20 the STUDENT file is OPENed for input. This causes the computer to search for the beginning of the file named STUDENT, so that it can be read later using INPUT statements. Files cannot be used unless they first are opened, so all programs will contain one OPEN statement for each file to be used. In the statement

```
OPEN 3, 1, 0, "STUDENT"
```

the 3 is called a *file number* (or *unit number*) and is used as an abbreviated name for the file throughout the rest of the program. It must always be used, and can be any number ranging from 1 to 255 inclusive. It is not

permanently associated with the file, and can be a different number each time the file is used. The 1 is the device number (cassette) and the 0 indicates it will be used for input.

Line 35 contains an INPUT# statement similar to the INPUT statement used in other examples. The difference is that the file number is used to indicate that input is to be obtained from the STUDENT file instead of the keyboard. Also a comma must follow this number.

Line 40 is used to cause the loop to be terminated. Each time through the loop one record is read from the STUDENT file. This record is always printed unless the student number is 9999, which indicates the sentinel record. Thus the loop is repeated 20 times, with 20 records read sequentially; the first 19 of these are used to print the students' names. This process is referred to as *sequential* reading of the file.

Line 60 causes the STUDENT file to be CLOSEd. It is always necessary to close a file when the program has finished processing it.

Notes:

(1) The sequence of items in the OPEN statement must be as indicated. Thus, it is not correct to write

```
OPEN 3, "STUDENT", 1,0
```

(2) The INPUT statement normally contains exactly the number and type of list items to match the fields in each record in the file. However the statement

```
INPUT#3, NUMBER,NAME$
```

would cause a record in the STUDENT file to be read, with values assigned to NUMBER and NAME$. The rest of the fields would be ignored.

(3) It is not possible to OPEN a file with the same unit number when it is already opened. This is one of the reasons the CLOSE statement is needed. If the CLOSE is omitted in error, the program will function but the file will remain open. If you have forgotten to close a file, you can do so using an immediate command. Simply type the CLOSE without a statement number.

8.5 Creating a File

The reader may be wondering what process was used to create the STUDENT file in the beginning. Example 8.2 is a program which will create a file named TELEPHON which contains names and telephone numbers of as many persons as you wish.

```
100 REM EXAMPLE 8.2
105 :
130 OPEN 6,1,2,"TELEPHON"
135 :
150 LOOP
155   PRINT "ENTER NAME AND PHONE"
160   INPUT NAME$,NUMBER$
170   IF NAME$="ZZZZ" THEN QUIT
180   PRINT#6,NAME$;",";NUMBER$
190 ENDLOOP
200 PRINT#6,"ZZZZ,999-9999"
220 CLOSE 6
230 STOP
```

The most important observation to make is that the name and telephone number are written on the file separated by a comma. This is accomplished by including the string

```
","
```

in the output list between NAME$ and NUMBER$. This must be done to provide the field separator in the output file.

NOTES: 1. The 2 in the OPEN statement causes the file named TELEPHON to be initialized for output.

2. The PRINT# statement is used to write records to the file. The file number 6 is used to indicate writing to the TELEPHON file instead of the screen.

3. The loop will terminate when the sentinel name ZZZZ is read. Note that you must also have typed in a sentinel telephone number as well; otherwise the input list will not be satisfied.

4. After the loop is terminated, a sentinel record is written in line 200. This is done so that end-of-file can be recognized during later use of the file.

8.6 Summary

This chapter has been included to introduce the reader to the fundamental concepts of input-output using cassette files. The subject of file processing is a fairly large one, and complete details are beyond the scope of this text. However, some simple disk and printer operations are described in the Appendices. For a complete description of the input-output facilities which are available, the reader is referred to the Reference Manuals which are supplied with specific devices.

8.7 Exercises

8.1 The file called STUDENT must be created before it can be used. The object of this exercise is to write a program which prompts the user for the appropriate data and creates the file shown in Figure 8.1. The program can be written to contain a loop as follows:

```
LOOP
  INPUT X$
  IF X$ = "QUIT" THEN QUIT
  PRINT#2,  X$
ENDLOOP
```

Each time through the loop a single record is entered as a string whose value is assigned to X$. Be sure to include this string in quotes as there are commas in the record. The value of X$ is then written onto the file whose unit number is 2, thus creating the record in the proper format.

Write the rest of the program, run it to create the file, and verify that the file contents are correct by displaying the entire file.

8.2 Write a program which reads the file STUDENT and displays the names of the male students.

8.3 Write a program which reads the file called STUDENT and calculates the class average for each of the five courses.

Chapter 9

Selection

In most programs the instructions to be executed will vary depending on the data being processed. For example in Chapter 3 a program was introduced which converted Celsius to Fahrenheit, or vice versa, depending on the value of a code which is typed. This selection is accomplished using a set of BASIC statements, namely IF, ELSE, ENDIF and ELSEIF. This chapter will discuss these statements and their application.

9.1 Termination of Loops

Virtually every program contains one or more loops. Every loop must be terminated, and all examples to this point have accomplished this using a special IF statement. For example, the statement

```
IF NAME$ = "ZZZZ" THEN QUIT
```

is used to terminate the loop in Example 8.2. This IF statement is referred to as the IF-THEN-QUIT combination as the keywords IF, THEN and QUIT are necessary. It is used in this text only for terminating loops and always has the format

```
IF relational expression THEN QUIT
```

If the relational expression is "true" the loop is terminated; if it is false the next instruction is executed and the loop continues.

The IF-THEN-QUIT combination is therefore a special-purpose mechanism. This chapter discusses another type of IF combination which is more generally applicable.

9.2 The IF-ENDIF Combination

Consider the program illustrated in Example 9.1.

```
100 REM EXAMPLE 9.1
105 :
110 OPEN 8,1,0,"STUDENT"
115 :
120 FSUM=0
125 FEMS=0
130 :
135 LOOP
140   INPUT#8,NO,N$,S$,A,M1
145   IF NO=9999 THEN QUIT
150   IF S$="F"
155     FEMS=FEMS+1
160     FSUM=FSUM+M1
165   ENDIF
170   PRINT NO,N$,S$,M1
175 ENDLOOP
180 :
185 FAVE=FSUM/FEMS
190 :
195 PRINT "FEMALE AVERAGE IS ";FAVE
200 CLOSE 8
205 :
210 STOP
```

The IF used on line 150 is followed by the relational expression S$ = "F". If this expression is "true" the block of statements between the IF and ENDIF are executed, namely statements 155 and 160. If the relational expression yields a value which is false, this block of two statements is not executed.

The effect in Example 9.1 is to calculate the aggregate algebra mark for females only, as well as the total number of females. When the loop has terminated the average algebra mark for females is computed and printed. The number, name, sex and algebra mark of every student is printed, as statement 170 is not in the range of the IF-ENDIF block.

To summarize, the IF and ENDIF statements are meant to work together to define a group or block of statements which may or may not be executed,

depending on the value of the relational expression associated with the IF statement. The statements in the block are sometimes referred to as the *range* of the IF-ENDIF combination.

NOTE: Only the first five fields in each record are read in line 140. The others are automatically skipped, as they are not included in the list of the INPUT statement.

9.3 The IF-ELSE-ENDIF Combination

```
110 REM EXAMPLE 9.2
115 :
120 OPEN 8,1,0,"STUDENT"
125 :
130 FSUM=0
135 MSUM=0
140 FEMS=0
145 MALES=0
148 :
150 LOOP
160   INPUT#8,NO,N$,S$,A,M1
170   IF NO=9999 THEN QUIT
180   IF S$="F"
190     FEMS=FEMS+1
200     FSUM=FSUM+M1
210   ELSE
212     MALES=MALES+1
214     MSUM=MSUM+M1
216   ENDIF
220   PRINT NO,N$,S$,M1
230 ENDLOOP
235 :
240 FAVE=FSUM/FEMS
245 MAVE=MSUM/MALES
247 :
250 PRINT "FEMALE AVERAGE IS ";FAVE
255 PRINT "MALE AVERAGE IS   ";MAVE
257 :
260 CLOSE 8
265 :
270 STOP
```

Example 9.2 is a slight variation of Example 9.1. Here the algebra average is calculated for males as well as females. This is accomplished by incorporating the ELSE statement within the range of the IF-ENDIF. This separates the range of statements into two blocks or sub-ranges, namely the statements between the IF and the ELSE, and the ones between the ELSE and the ENDIF. The former block is referred to as the *true range* and the latter as the *false range*. When the relational expression (which must always follow the IF keyword) is true, the statements in the true range are executed; if it is false those in the false range are executed. Thus the computer is able to select between two choices of action, depending on the current value of S$.

9.4 The IF-ELSEIF-ELSE-ENDIF Combination

Suppose it is required to count the number of students in the various age categories, namely ages 15, 16, 17 and 18. Example 9.3 is a program which accomplishes this using another BASIC statement, namely ELSEIF.


```
110 REM EXAMPLE 9.3
115 :
120 OPEN 8,1,0,"STUDENT"
125 :
130 N5=0
140 N6=0
150 N7=0
160 N8=0
165 :
170 LOOP
180   INPUT#8,NO,N$,S$,A
190   IF NO=9999 THEN QUIT
200   IF A=15
210     N5=N5+1
220   ELSEIF A=16
230     N6=N6+1
240   ELSEIF A=17
250     N7=N7+1
260   ELSEIF A=18
270     N8=N8+1
280   ELSE
290     PRINT "BAD RECORD"
300   ENDIF
310   PRINT NO,N$,S$,A
320 ENDLOOP
322 :
324 PRINT ""
325 PRINT "TOT15","TOT16","TOT17","TOT18"
330 PRINT N5,N6,N7,N8
335 :
340 CLOSE 8
345 :
350 STOP
```

In this example, the range of the IF (the statements between the IF and ENDIF) is separated into five sub-ranges. The first sub-range, contained between the IF and the first ELSEIF, is executed only if A = 15. The next sub-range, contained between the first and second ELSEIFs, is executed if A = 16. This pattern follows until all the ages are considered, providing one block of code for each age. The last block of code, between the ELSE and ENDIF will be executed only if none of the others is selected. This prints an error message because the file should contain only ages 15 to 18 inclusive.

This combination of statements is sometimes referred to as a *case construct* because only one of several "cases" is selected for processing.

9.5 General Rules Concerning IF-Statements

(1) Every IF statement contains a relational expression following the keyword IF.

(2) Every IF-ENDIF combination ends with the ENDIF statement. All statements between the IF and ENDIF are referred to as the range of the IF, regardless of which of the three IF-ENDIF combinations is being considered.

(3) Each of the statements in the range of the IF can be any statement. This means that the programmer can use other IF-ENDIF combinations *within* the range of an IF. These are referred to as *nested* IF's, and BASIC allows complete flexibility to nest IF's of all combinations, to any depth of complexity. It should be pointed out that such nesting makes the program difficult to read and it is generally advisable to avoid this type of programming if at all possible.

(4) The indentation of statements is used only to make the program more readable.

(5) When an IF-ENDIF combination has selected and executed the appropriate block of code, the processor always proceeds to the statement following the ENDIF statement associated with that IF combination.

(6) The IF-THEN-QUIT combination discussed at the beginning of the chapter, and used extensively to terminate loops, can also be used to terminate IF-ENDIF combinations. For example, if this statement is executed within any of the sub-ranges or blocks within the range of any IF-ENDIF combination, and if its relation evaluates as "true", the next statement to be executed will be the one immediately following the ENDIF statement associated with that combination. In other words the processing will exit from the range of the IF, just as it exited from the range of the LOOP in the case of termination of loops.

9.6 Exercises

9.1 a) Using the file called STUDENT as input, list the names of all students who have a mark of 80 or more in at least one course.

b) Using the file called STUDENT as input, list the names of all students who have a mark of 80 or more in at least two courses.

9.2 Using the file called STUDENT as input, list the name and marks for each of the nineteen students. Each mark is to have an asterisk (*) beside it, if the mark exceeds 79.

9.3 Using the file called STUDENT as input, list the name and a letter grade for each of the nineteen students. The grade is A if the average mark is 80 or above. The grade is B if the average mark is between 70 and 79 inclusive; otherwise the grade is Z.

9.4 Write a program which reads a positive integer number from the keyboard. This number is to be printed using 12 positions, filling all high-order positions with asterisks (*) as follows:

If the input is 123 print	*********123
If the input is 12345 print	*******12345
etc.	

Chapter 10

Repetition

In virtually every BASIC program it is necessary to repeat blocks or groups of statements. This has been illustrated in many examples using the LOOP-ENDLOOP combination of statements. This chapter summarizes some of the rules for loops, and introduces several other looping statements in BASIC.

10.1 Simple Loops

Suppose it is required to calculate the sum of the digits in a given integer. For example, if the integer is 2749, the sum of the digits is

```
2 + 7 + 4 + 9 = 22
```

This type of computation is often useful for calculating a check digit to be appended to a part number, student number, etc.

```
10 REM EXAMPLE 10.1
15 :
20 INPUT X$
25 :
30 L = LEN(X$)
35 I = 1
45 SUM = 0
50 :
55 LOOP
60   IF I>L THEN QUIT
70   SUM = SUM + VAL(MID$(X$,I,1))
75   I = I + 1
80 ENDLOOP
85 :
90 PRINT SUM
95 :
99 STOP
```

Example 10.1 is a program which reads a string of digits from the keyboard and assigns it to a string variable X$. (We use a string variable in order to be able to use string functions to simplify the computations.) The length of the string is determined, and a loop is repeated L times to add up the digits, one digit for each time through the loop. Finally the sum is printed.

The group of statements between the LOOP and ENDLOOP statements is said to be a block which is the *range* of the loop. All statements contained in this block are said to be "within" the loop, or "contained in" the loop.

Provision for the loop to be *terminated* is accomplished with the IF-THEN-QUIT combination. Note, that since this statement is placed at the beginning of the range, the other statements in the loop will not be executed if the null string is entered at the keyboard.

The loop is *initialized* by reading in the number, determining its length and setting I and SUM to their initial values.

Notes:

(1) Normally much of the data to be used in a loop must be initialized before entering the loop.

(2) The statements between any LOOP and ENDLOOP statements will be repeated endlessly, unless some mechanism such as the IF-THEN-QUIT causes termination. This mechanism can occur anywhere within the simple loop, to recognize the condition at the appropriate time.

(3) All statements in the range of the loop are indented for readability.

10.2 Nested Loops

```
10 REM EXAMPLE 10.2
12 :
15 LOOP
20   INPUT X$
25   IF X$ = "QUIT" THEN QUIT
30   L = LEN(X$)
35   I = 1
45   SUM = 0
55   LOOP
60     IF I>L THEN QUIT
70     SUM = SUM + VAL(MID$(X$,I,1))
75     I = I + 1
80   ENDLOOP
90   PRINT SUM
92 ENDLOOP
95 :
99 STOP
```

Example 10.2 is similar to Example 10.1 except that several numbers can be read one at a time, with the sum of the digits computed and printed. The program is terminated when the string "QUIT" is entered at the keyboard.

The reader will notice that this example contains a loop within a loop. The inside loop is said to be a *nested* loop, because it is contained in the range of the outside loop. This inside loop has its own range, which is a subrange of the range of the outside loop. The statements in the inner loop are further indented to make it easier to read the program.

The range of the inside loop is a separate block of statements which is part of the outside loop but is considered to be at a lower level than

the other statements in the outside loop. When the inside loop terminates, the rest of the statements in the outside loop continue to function until this outside loop also terminates.

Another way of stating the previous paragraph is to say that the statement

```
IF I > L THEN QUIT
```

makes provision for only the inside loop to terminate. After it terminates, the statement

```
PRINT SUM
```

is executed and then the outside loop is repeated. This outside loop is finally terminated when the string QUIT has been entered, and the statement

```
IF X$ = "QUIT" THEN QUIT
```

is executed.

Notes:

(1) Each loop usually has its own mechanism for termination.

(2) Each loop usually has some initialization statements. In this example, the outside loop has no explicit initialization statements.

(3) The loops can be nested to any desired level. However, such nesting can make the program more difficult to read and debug. When several levels of looping are required it is usually advisable to use Procedures (Chapter 13) to repackage the program to improve readability.

(4) Because of the indentation, it is clear which ENDLOOP statement belongs to each of the LOOP statements; however the rule is as follows:

a) Find all LOOP-ENDLOOP combinations which have no intervening LOOP or ENDLOOP statements in their range. These particular combinations are said to be *correctly paired*, or more briefly, paired.

b) Find all LOOP-ENDLOOP combinations which have no intervening unpaired LOOP or ENDLOOP statements in their range. These are then paired.

c) Step b) is repeated until all LOOP-ENDLOOP combinations have been paired.

d) If there are any LOOP or ENDLOOP statements remaining which have not been paired, the loops are incorrectly nested.

10.3 Loops with the WHILE-ENDLOOP Combination

```
10 REM EXAMPLE 10.3
15 :
20 INPUT X$
25 :
30 L = LEN(X$)
35 I = 1
45 SUM = 0
47 :
55 WHILE I <= L
70   SUM = SUM + VAL(MID$(X$,I,1))
75   I = I + 1
80 ENDLOOP
85 :
90 PRINT SUM
95 :
99 STOP
```

Example 10.3 is identical to Example 10.1 except that a new statement has been used, namely

```
WHILE I <= L
```

The WHILE statement can only be used to start a loop, and is an alternative to the LOOP statement. The keyword WHILE is always followed by a relational expression. This expression is always evalulated at the

beginning of the loop, and the statements in the range of the loop are executed only if its value is "true".

The WHILE statement provides no additional function for the programmer. It is never necessary, but is used to make the program more readable in some cases. It functions in precisely the same manner as the following two statements used in Example 10.1.

```
LOOP
  IF I > L THEN QUIT
```

10.4 Loops with the LOOP-UNTIL Combination

```
10 REM EXAMPLE 10.4
15 :
20 INPUT X$
25 :
30 L = LEN(X$)
35 I = 1
45 SUM = 0
47 :
55 LOOP
60   IF I>L THEN QUIT
70   SUM = SUM + VAL(MID$(X$,I,1))
75   I = I + 1
80 UNTIL I > 5
85 :
90 PRINT SUM
95 :
99 STOP
```

Example 10.4 is identical to Example 10.1 except for an extra provision. The sum consists only of the first 5 digits in the number. If the number has fewer than 5 digits, the sum includes all of the digits.

Here a new statement has been introduced, namely

```
UNTIL I > 5
```

The UNTIL statement can only be used to end a loop, and is an alternative to the ENDLOOP statement. The keyword is always followed by a relational expression. This expression is always evalulated at the end of the loop, and the loop is terminated if its value is "true".

Just as is the case with the WHILE statement, the UNTIL provides no additional function for the programmer. It is never necessary, but is used to make the program more readable under the appropriate circumstances. It functions in precisely the same manner as if the following two statements were used in its place.

```
  IF I > 5 THEN QUIT
ENDLOOP
```

10.5 Loops with the WHILE-UNTIL Combination

```
10 REM EXAMPLE 10.5
15 :
20 INPUT X$
25 :
30 L = LEN(X$)
35 I = 1
45 SUM = 0
47 :
55 WHILE I <= L
70   SUM = SUM + VAL(MID$(X$,I,1))
75   I = I + 1
80 UNTIL I > 5
85 :
90 PRINT SUM
95 :
99 STOP
```

Example 10.5 is another version of Example 10.4 which uses the WHILE at the beginning and the UNTIL at the end of the loop. Note that the WHILE relational expression is evaluated *before* each iteration of the loop; the UNTIL relational expression is evaluated *after* each iteration of the loop.

10.6 Loops with the FOR-NEXT Combination

Suppose it is required to calculate the sum of the integers from 1 to 8. The computation to be performed is

```
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8
```

Example 10.6 is an illustration of a program which uses the FOR-NEXT combination of statements for solving this simple problem.

```
10 REM EXAMPLE 10.6
15 :
25 SUM = 0
27 :
30 FOR I = 1 TO 8
35   SUM = SUM + I
40 NEXT I
45 :
50 PRINT SUM
55 :
60 STOP
```

The statements in the loop are contained between the FOR and the NEXT statements. In this case the range of the loop consists of only one statement, namely

```
SUM = SUM + I
```

The loop is repeated for all integer values beginning with 1, up to and including 8.

The FOR statement always contains an *index-variable*, in this case I. This variable is set to the *initial value* (in this case 1) and the loop is executed the first time.

The end of the loop is specified with the NEXT statement. This keyword is always followed by the index-variable name, in this case I. When this statement is executed the value of the index-variable I is increased by one. Then the index is checked to be sure its value is not greater than the *terminal value* (in this case 8). If the index value satisfies this test, control returns to the FOR statement and the loop is executed again.

NOTE: When the FOR loop is completed I has a value of 9.

Example 10.7 calculates the sum of the odd integers from 1 to 99 inclusive. Notice we have introduced the STEP keyword in the FOR statement. This causes the NEXT statement to increment I by two rather than one.

```
10 REM EXAMPLE 10.7
15 :
25 SUM = 0
27 :
30 FOR I = 1 TO 99 STEP 2
35   SUM = SUM + I
40 NEXT I
45 :
50 PRINT SUM
55 :
60 STOP
```

Example 10.8 illustrates the nested FOR loop. Here a table of the sum of the integers from 1 to N, for all N ranging from 5 to 20 is calculated and printed. Note that the terminal value of the index-variable for the inner loop is itself a variable, namely J.

```
10 REM EXAMPLE 10.8
15 :
25 FOR J = 5 TO 20
30   SUM = 0
35   FOR I = 1 TO J
40     SUM = SUM + I
45   NEXT I
50   PRINT J,SUM
55 NEXT J
57 :
75 STOP
```

Notes:

(1) The FOR statement has the general form

```
FOR index-variable = A TO B STEP C
```

A, B, and C are all numeric expressions, and their values can be any real constants. Thus it is permissible to have a statement such as

```
FOR K = 63.5 TO 21.5 STEP -.5
```

which causes K to begin with a value of 63.5, and be reduced by .5 each time through the loop, with a terminal value of 21.5.

(2) The STEP keyword can be used with a negative value. In this case the value of the index-variable will be decreased when the NEXT statement is encountered. Looping will terminate when the value of the index-variable becomes less than the terminal value.

(3) Statements in a FOR loop are always executed once, even though one might expect otherwise. For example in the statement

```
FOR I = 6 TO 4 STEP 2
```

the increment is 2 and therefore positive. When executed for the first time the index-variable I is assigned the value 6. This already exceeds the terminal value 4, so you might expect the FOR loop to terminate immediately; however it is always executed once since the test is performed when the NEXT statement is encountered.

(4) Any statement can be placed within the range of the FOR loop.

(5) A FOR loop *cannot* be terminated prematurely using the IF-THEN-QUIT combination.

10.7 Summary

The only looping mechanism actually required to solve any computing problem is the LOOP-ENDLOOP with IF-THEN-QUIT, used in previous chapters. The other statements introduced in this chapter are an enrichment providing convenience to the programmer, and enhanced readability in appropriate circumstances.

10.8 Exercises

10.1 Write a program which uses the file STUDENT as input and prints each record in approximately the following format:

e.g.

```
110      STEVENS
                    65
                    63
                    85
                    56
                    76
                    --
AVERAGE  69
```

10.2 Write a program which uses the file STUDENT as input and prints the names of the students, with three names on each line. As there are 19 students, the last line will have only one name.

10.3 Write a program which prints a calendar page for a given month. The program accepts as input the name of the month, number of days in the month, and the day of the week on which the month begins.

10.4 a) Write a program which prints a triangle of asterisks (*) which is similar to the following:

```
*
**
***
****
```

The input to the program is the size of the triangle, in this case 4, which is the number of rows in the vertical side of the triangle.

b) Write a program which produces triangles which are similar to the following:

```
   *
  **
 ***
****
```

c) Write a program which produces triangles which are similar to the following:

```
* * * *
 * * *
  * *
   *
```

Chapter 11

Relational Expressions

Relational expressions have been introduced informally in earlier chapters. The purpose of this chapter is to give further examples and introduce the logical operators AND, OR, and NOT.

11.1 Simple Relational Expressions

Relational expressions are used in IF, ELSEIF, WHILE and UNTIL statements, and have been included in most programs in this book. Examples are:

```
(i)    SEX$ = "F"
(ii)   NUMBER = 9999
(iii)  NUMBER < 0
(iv)   AGE <> 19
(v)    16 > 14
(vi)   "CAT" = " CAT"
(vii)  "CAT" = "CAT "
(viii) "CAT" < "DOG"
```

Each of these expressions has a value "true" or "false". In example (i) the current value assigned to the string variable SEX$ is compared to the string constant "F". If they are identical, the expression has a value of true, otherwise it is false.

In Example (iii) the expression will be true only if the current value of NUMBER is negative.

Example (iv) illustrates the use of the symbols <> to mean "not equal to". Thus if AGE is not equal to 19, the expression is true.

Example (v) is a special case because no variables are involved. Since 16 is greater than 14 the expression is always true.

Example (vi) illustrates an important point. The two string constants "CAT" and " CAT" are not equal because the latter has 4 characters beginning with a blank. It is always good policy when comparing strings to have them the same size, with leading blank spaces removed. It is also true that "CAT" and "CAT " (example vii) are not equal, even though the blank is on the right. For two strings to be equal they must be of the same length. In Example (viii) the expression is true because strings can be compared, and the usual ordering between letters of the alphabet applies here.

The reader may be wondering why examples (i) and (ii) are not assignment statements. In fact, they could be assignment statements, but for purposes of this discussion they are assumed to be the relational expression in an IF, ENDIF, WHILE, or UNTIL statement. The computer detects that they are relational expressions by examining the *context* in which they are used.

11.2 Relational Operators

As has been mentioned in Chapter 1, there are six relational operators as follows:

=	equals
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
<>	not equal

These operators are *binary*, which means that there are always two operands, one on each side of the operator. Thus in example (iv) of the previous section, AGE and 19 are operands and <> is the operator. The two operands are always of the same kind, either string expressions or numeric expressions. It would make no sense to try to compare a string of characters to a number. When the operands are expressions they are evaluated before the relational expression is computed. Thus with the relational expression

```
AGE-14 >= TEST+1
```


the two numeric expressions AGE-14 and TEST+1 are computed and then the relational operator >= is applied to the two values, yielding a result of either true or false.

11.3 Compound Relational Expressions

Relational expressions can be combined using the logical operators AND, OR, and NOT to produce other relational expressions. Consider the following examples:

```
(i)    (SEX$ = "F") AND (AGE > 17)

(ii)   (SEX$ = "F") AND ((AGE = 17) OR (AGE < 10))

(iii)  NOT (SEX$ = "F")
```

The compound relation in example (i) will be true only if SEX$ = "F" and AGE > 17 are both true.

In example (ii) the sub-expression

```
(AGE = 17) OR (AGE < 10)
```

is evaluated first, because this expression is enclosed in parentheses. The sub-expression result will be true if either of the relational expressions has a true value. This value is then used with the result of sub-expression SEX$ = "F" to produce "true" only if both values are true.

In example (iii) the NOT is used to reverse the value of SEX$ = "F". If SEX$ = "F" is true, then NOT(SEX = "F") is false, and vice versa.

11.4 Logical Operators

The three logical operators are defined as follows:

(1) AND is a binary operator whose two operands are relational expressions. If the two expressions are true, the resulting operation is true, otherwise it is false.

(2) OR is a binary operator whose two operands are relational expressions. If either of the two expressions is true the result of the operation is true, otherwise it is false.

(3) NOT is a unary operator whose operand is a relational expression. If the value of the operand is true, the result of the operation is false; if the value of the operand is false, the result is true.

The operators have the following priority with respect to each other:

```
NOT
AND and OR
```

For example, a NOT operation takes precedence over an AND or OR, provided there are no parentheses. Of course, the usual conventions with respect to parentheses are used, with the expression in the innermost parentheses being evaluated first.

Three levels of operators have been described in this text. In order of decreasing priority they are

(i) arithmetic operators or string operators
(ii) relational operators
(iii) logical operators

Therefore in an example such as

```
NOT X + 3 > 6
```

the numeric expression X+3 is calculated. Then the relational expression X+3 > 6 is evaluated. Finally the result is operated upon with the logical operator NOT. Incidentally the whole expression should have been written with parentheses for readability, as follows

```
NOT((X+3) > 6)
```

11.5 Summary

Relational expressions play an indispensable role in most computer programs. The logical operators AND, OR, and NOT provide considerable flexibility in the construction of compound relational expressions. However,

the reader should be warned that large, complex relational expressions are difficult to understand and make debugging difficult. Therefore they should be used with discretion. Clarity is often introduced by using parentheses, even when these are not strictly required because of the priority rules.

11.6 Exercises

11.1 Using the file STUDENT as input, write a program which prints the names of all the male students who are age 17 and have received a mark of less than 80 in algebra.

11.2 Using the file STUDENT as input, write a program which prints the names of the male students who are not age 17, and who have an average mark exceeding 79.

11.3 Using the file STUDENT as input, write a program which prints the names and marks of both the male and female students with the highest average mark.

11.4 Using the file STUDENT as input, write a program which prints the names and marks of both the male and female students with the highest average mark, and who are sixteen years of age and under.

Chapter 12

Tables

Many applications require the use of tables or matrices (also called arrays or vectors) to store data in a convenient form. This chapter introduces several features which make this type of processing convenient to program.

The various features are illustrated with a series of examples to demonstrate the concepts.

12.1 Example 12.1

Suppose it is required to read a list of words into the computer and print the entire list in reverse order of input. All of the words would have to be stored in a *table* in the computer, because the first output line cannot be printed until the last word has been read.

A table of variables can be formed using the DIM statement, as in Example 12.1.

```
10 REM EXAMPLE 12.1
15 :
40 DIM NAME$(4)
45 :
50 PRINT "TYPE FIRST NAME"
60 INPUT NAME$(1)
62 PRINT "TYPE SECOND NAME"
64 INPUT NAME$(2)
66 PRINT "TYPE THIRD NAME"
68 INPUT NAME$(3)
70 PRINT "TYPE FOURTH NAME"
72 INPUT NAME$(4)
74 PRINT "NAMES IN REVERSE ARE"
76 PRINT NAME$(4)
78 PRINT NAME$(3)
80 PRINT NAME$(2)
82 PRINT NAME$(1)
85 :
90 STOP
```

Statement 40 uses the DIM keyword to declare a table which has five *entries* or *elements*. These elements are variables which can be individually referenced as NAME$(0), NAME$(1), NAME$(2), NAME$(3) AND NAME$(4). The integer contained in parentheses is often called a *subscript*. It is also referred to as the *index* of the table entry.

Example 12.1 is designed to read four names and store them in turn in four elements of the table namely NAME$(1), NAME$(2), NAME$(3) and NAME$(4). Then the elements are selected in reverse order, and printed.

Notes:

(1) The table contains five elements which have indices 0, 1, 2, 3, and 4. The *range* of indices is said to be 0 to 4 inclusive. Example 12.1 does not use the entry with index 0.

(2) The table defined in the DIM statement can have as many elements as required.

(3) If several tables are needed they can be declared in separate DIM statements; alternatively one DIM statement can be used as follows:

```
DIM NAME$(4), MARK(5)
```

(4) A table can have elements which are string variables or numeric variables, but not both.

(5) The index or subscript is an integer value which must be within the appropriate range. If the index is a non-integer value, the BASIC system will automatically convert it to an integer using rounding.

(6) The quantities within parentheses in a DIM statement denote the index of the last entry in the tables. These quantities are always positive integers or numeric variables.

(7) DIM statements are usually placed near the beginning of the program to improve readability. In any case they must appear prior to the use of one of the table elements being defined.

(8) In constructing the names for tables, we obey the same rules as for regular variables.

12.2 Example 12.2

Example 12.1 is obviously a very cumbersome way of writing the program, especially if a large table were involved. Example 12.2 is an improved version which takes fewer statements to write, and produces similar output.

```
10 REM EXAMPLE 12.2
15 :
20 DIM NAME$(4)
25 :
30 FOR I = 1 TO 4
40   PRINT "NAME PLEASE"
50   INPUT NAME$(I)
60 NEXT I
65 :
70 PRINT "NAMES IN REVERSE ARE"
75 :
80 FOR I = 4 TO 1 STEP -1
85   PRINT NAME$(I)
90 NEXT I
95 :
99 STOP
```

In this example the index of the table NAME$ is a variable I which is assigned appropriate values as the loops are executed. In fact, this index can be any numeric expression.

12.3 Example 12.3

```
100 REM EXAMPLE 12.3
105 :
130 OPEN 6,1,0,"STUDENT"
135 :
140 DIM MK(5)
145 :
150 LOOP
160   INPUT#6,NO,N$,S$,A,MK(1),MK(2),MK(3),MK(4),MK(5)
170   IF NO=9999 THEN QUIT
180   SUM=0
190   FOR I = 1 TO 5
200     SUM = SUM + MK(I)
210   NEXT I
220   AVE=SUM/5
230   PRINT NO;N$;AVE
240 ENDLOOP
245 :
250 CLOSE 6
255 :
260 STOP
```

In this example all records in the STUDENT file are read in turn. For each record the 5 marks are read into a table called MARK, and the average is computed; then the student's number, name, and average is printed.

The output appears on the screen as follows:

```
110 STEVENS 69
297 WAGNER 78.8
317 RANCOURT 70
364 WAGNER 73
617 HAROLD 78.8
998 WEICKLER 74.2
1203 WILLS 74.8
1232 ROTH 71.6
1234 GEORGE 67.6
1265 MAJOR 67
1568 POLLOCK 85.2
1587 PEARSON 55
1617 REITER 80.2
2028 SCHULTZ 67.8
2036 BROOKS 67.4
2039 ELLIS 85
2049 BECKER 66.4
2055 ASSLEY 63.2
2087 STECKLEY 70
```

12.4 Example 12.4

```
100 REM EXAMPLE 12.4
105 :
140 DIM X(100)
145 :
150 J = 0
155 :
160 LOOP
170   PRINT "NUMBER PLEASE"
180   INPUT NUMBER
190   IF NUMBER < 0 THEN QUIT
200   J = J + 1
210   X(J) = NUMBER
220 ENDLOOP
225 :
230 SUM = 0
235 :
240 FOR I = 1 TO J
250   SUM = SUM + X(I)
260 NEXT I
265 :
270 AVE = SUM/J
275 :
280 PRINT "THESE ARE ABOVE AVE"
285 :
290 FOR I = 1 TO J
300   IF X(I) > AVE
310     PRINT X(I)
320   ENDIF
330 NEXT I
335 :
340 STOP
```

In this example a table X of 101 entries is defined. Then up to 100 positive numbers are read from the keyboard and stored in the table. The average is computed and then all entries whose value exceed this average are printed. Note that the variable J is used to store the current size of the table. Even though the table can have up to 101 entries, only J of them are used when the program is run.

12.5 Example 12.5

```
100 REM EXAMPLE 12.5
110 :
130 OPEN 6,1,0,"STUDENT"
135 :
140 DIM AGESUM(4)
145 :
150 FOR I = 1 TO 4
160   AGESUM(I) = 0
170 NEXT I
175 :
180 LOOP
190   INPUT#6,NO,N$,S$,A
200   IF NO = 9999 THEN QUIT
210   INDEX = A - 14
220   AGESUM(INDEX) = AGESUM(INDEX)+1
230 ENDLOOP
235 :
240 PRINT "AGE","COUNT"
245 :
250 FOR I = 1 TO 4
260   PRINT I+14,AGESUM(I)
270 NEXT I
275 :
280 CLOSE 6
285 :
290 STOP
```

Suppose it is desired to count the number of students in the STUDENT file for each of the ages 15, 16, 17, and 18. In Example 12.5 the table AGESUM is defined to accumulate the four totals as the computation proceeds. Thus AGESUM(1) will contain a running total for all students who are age 15; AGESUM(2) will be used for age 16, AGESUM(3) for age 17 and AGESUM(4) for age 18. In line 210 the INDEX of the table is computed by subtracting 14 from A. Thus the ages 15, 16, 17, 18 are "translated" to index values 1, 2, 3, 4 respectively so they will be in the proper range.

12.6 Two Dimensional Tables

Sometimes it is useful to have tables which contain *rows* and *columns*. For example the following table indicates the number of females and males in each of the four age groups for records in the STUDENT FILE

AGE	FEMALES	MALES
15	2	2
16	2	4
17	0	6
18	0	3

This table has four rows and two columns, and contains 8 entries. Example 12.6 is a program which produces this table.

```
100 REM EXAMPLE 12.6
105 :
130 OPEN 6,1,0,"STUDENT"
135 :
140 DIM COUNT(4,2)
145 :
150 FOR I = 1 TO 4
160   FOR J = 1 TO 2
170     COUNT(I,J) = 0
180   NEXT J
190 NEXT I
195 :
200 LOOP
210   INPUT#6,NO,N$,S$,A
220   IF NO = 9999 THEN QUIT
230   AGE = A - 14
240   IF S$ = "F"
250     SEX = 1
260   ELSE
270     SEX = 2
280   ENDIF
290   COUNT(AGE,SEX)=COUNT(AGE,SEX)+1
300 ENDLOOP
305 :
310 PRINT "AGE","FEMALES","MALES"
315 :
320 FOR I = 1 TO 4
330   PRINT I+14,COUNT(I,1),COUNT(I,2)
340 NEXT I
345 :
350 CLOSE 6
355 :
360 STOP
```

The two-dimensional table COUNT is defined in line 140 as follows

```
140 DIM COUNT(4,2)
```

Eight entries are referenced in the program using two indices, which are called AGE and SEX. The AGE index is calculated in line 230 and will have a value between 1 and 4 inclusive. In the IF block between lines 240 and 280 inclusive, the sex value "F" is translated to 1 and the sex value of "M" to 2, thus calculating the SEX index in the proper range of 1 or 2.

When the table is printed beginning in line 320 the output is produced as 4 lines, one line for each row. Each row contains three numbers, one for age and one for each sex.

The table COUNT actually has 15 entries since the first subscript can take the values 0 through 4 and the second subscript 0 through 2. We did not use the value 0 for either subscript in this example.

12.7 Multi-Dimensional Tables

The BASIC system permits the use of tables with an arbitrary number of dimensions. While very large dimensions are seldom practical it is useful to have the facility available if needed. To declare a table of 5 dimensions you could write

```
DIM TABLE (20, 40, 6, 8, 2)
```

This will create a table of 21 x 41 x 7 x 9 x 3 = 162729 entries! When using such large tables remember that the computer is of limited size, and you could easily exhaust the space available.

12.8 Summary

In many practical programming situations, one or more tables will be necessary. It is therefore important to learn this facility thoroughly. The following exercises will provide the necessary practice.

12.9 Exercises

12.1 Write a program which reads the 19 records in the file STUDENT and stores them in a table. The program then determines the student with the highest mark in algebra and prints that record.

12.2 a) Write a program which reads several words from the keyboard and stores them in a table. The program then chooses the smallest word in the alphabetic sequence and prints it.

b) Expand on the program so that each of the words is printed in sequence, thus producing an alphabetic listing of all the words.

c) Use the program in b) to enter each of the characters on the key-board as single-character "words". In this way a complete listing of all of the available characters will be produced in "alphabetical order". This order is known as the *collating sequence* of the character set, and may be different for different types of computers.

12.3 A cryptogram is a common form of puzzle which presents a string of text (message) in a coded form to the reader. The object is to decode the message so it makes sense. In a cryptogram each letter is replaced by a different letter of the alphabet. For example the word CAT might be written as ZTB where C, A and T are encoded as Z, T and B respectively. This correspondence can be set up as a table within the computer.

a) Write a program which permits the user to input the characters as an encoding table. Then a sentence should be read from the keyboard, and the encoded sentence subsequently printed.

b) Modify the program so that the encoded sentence can be read from the keyboard to produce the original sentence.

12.4 Write a program which reads the 19 records in the file STUDENT and stores them in a table. The program then prints the entries in the table one at a time but in descending sequence by algebra mark, the highest algebra mark first. You might consider finding the entry containing the largest algebra mark, printing that entry, replacing it by zero and repeating the process for all entries in the table.

Chapter 13

Procedure Subprograms

In previous chapters there are a number of examples of built-in functions which have been provided as part of the BASIC system. This chapter explains how the programmer can construct other functional units called procedure subprograms which may be useful for the particular problem being considered. Procedures can be used as a means of packaging the program into meaningful units, making it more readable and easier to maintain.

13.1 An Introduction to Procedure Structures

The reader is requested to review Example 12.3 which is a program to read the STUDENT file, calculate the average of the five marks for each of the 19 students in the file, and print a listing of the student number, name and average for each student.

Example 13.1 is another version of the same program. It introduces three new keywords, namely CALL, PROC, and ENDPROC, located at lines 160, 600, and 660 respectively. The two keywords PROC and ENDPROC define a group of statements which is called a *procedure*. In the example, the procedure consists of the 7 statements between lines 600 and 660 inclusive. The *name* of the procedure is AVERAGE. The procedure name always follows the keyword PROC, and consists of as many letters or digits as desired; it must start with a letter, and all letters and digits are significant. That is, the procedure name CAT is different from the procedure name CA. In the example, the procedure calculates the average of the 5 marks, and assigns its value to AVE in line 650.

```
100 REM EXAMPLE 13.1
105 :
110 DIM M(5)
115 :
120 OPEN 6,1,0,"STUDENT"
125 :
130 LOOP
140   INPUT#6,NO,N$,S$,A,M(1),M(2),M(3),M(4),M(5)
150   IF NO = 9999 THEN QUIT
160   CALL AVERAGE
170   PRINT NO,N$,AVE
180 ENDLOOP
185 :
190 CLOSE 6
195 :
200 STOP
205 :
206 :
600 PROC AVERAGE
605 :
610 SUM = 0
620 FOR I = 1 TO 5
630   SUM = SUM + M(I)
640 NEXT I
645 :
650 AVE = SUM / 5
655 :
660 ENDPROC
```

The statements at the beginning of the program, namely those between line 100 and 200 inclusive, are referred to as the *mainline* program and the procedure is called a *subprogram.* When the program is RUN, it begins to execute at the first statement in the mainline program. When the statement in line 160 is executed, namely

CALL AVERAGE

the computer causes the statements in the procedure named AVERAGE to be executed, namely lines 600 to 660 inclusive. This assigns a value to the variable AVE. The mainline program resumes at line 170 after execution of the ENDPROC statement. The mainline program is said to have *called* the procedure subprogram.

The purpose of this example has been to describe how a procedure is defined, and explain the mechanics of its operation. In the following sections we explain some of the reasons procedures are used, and describe their other properties.

13.2 Segmenting a Program Using Procedures

Most real-life programs are quite large, and can contain hundreds or thousands of statements. The program becomes difficult to read and understand. One of the most important uses of procedure subprograms is to *package* the program into smaller segments, each of which performs a specific well-defined operation or function.

Example 13.2 is another version of Example 12.4 which demonstrates this "repackaging" of a program.

The program has been packaged into four components - the mainline program and three procedures.

(i) The procedure READNUMBERS causes several numbers to be read from the keyboard and stores them in a table called X. The value of the variable J will indicate the number of entries which have been read and stored in the table after the procedure has been called.

(ii) The procedure CALCAVERAGE uses entries in the table X, and the value of J, as data and calculates the average of the values in the table. This is assigned to the variable AVE.

(iii) The procedure PRINTNUMBERS uses the entries in the table, the value of J, and the value of AVE as input and prints a list of the entries which is above average.

(iv) The mainline program is used to call the three procedures, in the proper sequence. This happens when statements 150, 160 and 170 are executed. For example, when line 150 is executed, the procedure name READNUMBERS is encountered. This causes the statements numbered 500 to 590 to be executed. Similarly in line 160 and 170 the procedures CALCAVERAGE and PRINTNUMBERS are called.

```
100 REM EXAMPLE 13.2
105 :
140 DIM X(100)
145 :
150 CALL READNUMBERS
160 CALL CALCAVERAGE
170 CALL PRINTNUMBERS
180 STOP
186 :
500 PROC READNUMBERS
505 :
510 J = 0
520 LOOP
530   PRINT "NUMBER PLEASE"
540   INPUT NUMBER
550   IF NUMBER < 0 THEN QUIT
560   J = J + 1
570   X(J) = NUMBER
580 ENDLOOP
585 :
590 ENDPROC
596 :
800 PROC CALCAVERAGE
805 :
810 SUM = 0
820 FOR I = 1 TO J
830   SUM = SUM + X(I)
840 NEXT I
850 AVE = SUM / J
858 :
860 ENDPROC
865 :
900 PROC PRINTNUMBERS
902 :
905 PRINT "NUMBERS ABOVE AVG ARE"
910 FOR I = 1 TO J
920   IF X(I) > AVE
930     PRINT X(I)
940   ENDIF
950 NEXT I
955 :
960 ENDPROC
```


Notes:

(1) The line-number range for each of the three procedures has been chosen to start with an artificially high statement number. This is to separate the procedures from one another, and to permit the programmer to easily increase the number of statements within the procedure, if the need should arise.

(2) The procedures are separated from each other, and from the mainline by null statements. This is to cause a separation in the listing, thus making the program easier to read.

13.3 Some Rules about Procedures

1. All variables used in the two example programs have a *global definition.* For example, assume a variable X has been assigned a value 6 in the mainline program. When a procedure subprogram is executed, this variable can be used, and if desired it can be altered. As well, any variable introduced in the subprogram is "known", and therefore can be used by all other subprograms, as well as the mainline program.

2. In the examples, procedure subprograms have been called by the mainline program. In fact, procedure subprograms may also be called by other procedure subprograms.

3. A procedure always is terminated with an ENDPROC statement. The QUIT statement is never used to terminate a procedure.

13.4 Summary

Large programs should always be written as a collection of *modules*, each of which performs a specific task. The Waterloo BASIC facility to define procedures provides a useful mechanism for creating these modules.

13.5 Exercises

13.1 a) Please refer to Example 7.5 in Chapter 7. This program computes the number of occurrences of the letter A in a given sentence. Repackage this program so it becomes a procedure. The procedure should compute the number of occurrences.

b) Generalize the procedure in part a) so that it will determine the number of occurrences of any given character in any given string. Do this by assigning values for two variables ((i) number of occurrences and (ii) the string) prior to calling the procedure.

c) Use the procedure of part b) to write a program which computes the total number of vowels in a given sentence.

13.2 Write a procedure which determines the lowest entry in an array of string constants. Use this procedure to print a list of the names of students in the file called STUDENT, in alphabetical sequence.

13.3 a) In Exercise 12.4 the program repeatedly finds the entry with the highest algebra mark. Write this part of the program as a procedure subprogram which computes the position of the entry in the table.

b) Modify the procedure in a) so that it will work for any one of the marks in the entry.

Appendix A

BASIC Language

A program in the BASIC language is composed of statement lines. Each line consists of a *statement number* followed by one of the BASIC statements described in this Appendix. Successive lines in a BASIC program are arranged in ascending order by statement number.

Note:

(1) The statement number can be any positive integer in the range 1 through 63999.

(2) A statement line can be up to 80 characters long.

(3) Blank characters in a statement line are ignored by the computer unless they are part of a string constant. However, they can be used to improve the readability of the program.

Statement lines are normally entered directly from the keyboard. Entering a line with the same number as an existing line causes the existing line to be replaced by the new one. Entering a line number by itself, with no BASIC statement following, causes an existing line with that number to be deleted.

Almost all BASIC statements can be entered at the keyboard without the accompanying line number. This causes the statement entered to be executed immediately and is sometimes called "direct mode" or "immediate mode". The statement is not saved, however, and if it is to be

executed again, it must be re-entered. This is useful at times when debugging a program or doing simple calculations.

In the following descriptions, a string expression is an expression that produces a string; a numeric expression is an expression that produces a number.

A.1 Assignment Statement

Syntax:

```
LET variable = expression
or
variable = expression
```

Operation:

The "expression" may be a numeric or string expression. It is evaluated and the result is assigned to "variable". A numeric expression must be assigned to a numeric variable and a string expression must be assigned to a string variable.

Example:

```
10 LET X = 2+A*3
20 A$ = "ABC"+B$
```

A.2 CALL Statement

Syntax:

```
CALL procedure-name
```

Operation:

Execution of this statement causes control to transfer to the procedure with the specified "procedure-name". This procedure is defined by the PROC and ENDPROC statements. After the statements in the procedure are executed, control returns to the statement following the CALL statement.

Example:

```
10 CALL FACTORIAL
```

This statement causes the statements in the procedure FACTORIAL to be executed; then control returns to the statement following statement 10.

A.3 CLOSE Statement

Syntax:

```
CLOSE logical-file
```

Operation:

This statement releases a file previously opened with the OPEN statement. The logical file number is then available for use with another file.

A.4 CMD Statement

This statement is used to interface with external devices attached to the Commodore computer. Its use is beyond the scope of this text. Information on this statement can be found in the appropriate Commodore manuals.

A.5 CONT Statement

Syntax:

```
CONT
```

Operation:

This statement can only be issued in "direct mode". It allows the program to resume execution after it has been stopped by the Stop-key, or the execution of a STOP or END statement. This statement cannot be used after an error, any editing of the program, or after a NEW or CLR command has been used.

A.6 DATA Statement

Syntax:

```
DATA constant, constant,...
```

Operation:

The DATA statement defines data items to be read by READ statements. The "constant" can be numeric or string. If string constants are to contain colons (:), commas (,) or blanks, they must be enclosed in double-quotes ("). If the string contains a quote mark it must be a single quote (').

Example:

```
200 DATA 3.4, -1E4, "HELLO", IT'S, BOY
```

A.7 DEF Statement

Syntax:

```
DEF FNname(argument) = expression
```

Operation:

When this statement is encountered during execution, a user-created arithmetic function is defined with the name FNname. The function name must be composed of the letters FN followed by a valid arithmetic variable name. Only one argument is allowed which can be used as a dummy variable in the expression. Other variables from the program may be used in the function definition. User-defined string functions are not allowed.

Example:

```
10 DEF FNA(X) = 1/X
```

This statement defines a function FNA(X) which computes the inverse of X. Thus FNA(4) would have the value .25.

A.8 DIM Statement

Syntax:

```
DIM variable-name(dimension,...),...
```

Operation:

This statement causes an array with the name "variable-name" to be created containing a number of elements equal to the product of all its "dimensions", each incremented by 1. An array can have an arbitrary number of dimensions and a DIM statement can define an arbitrary number of arrays. All array elements are initialized to zero or the null string. Each dimension can be specified by either a constant or a variable, and defines a range of elements numbered from 0 to the specified dimension. If a variable is specified as a dimension, it must have been defined in a previously executed statement.

Example:

```
10 DIM A(40,2),B$(N)
```

defines two arrays. Array A contains (40+1)*(2+1) = 123 numeric elements. Array B$ contains N+1 string elements.

A.9 ELSE Statement

Syntax:

```
ELSE
```

Operation:

This statement must be used in conjunction with the IF and ENDIF statements. It specifies the beginning of a block of statements to be executed when all previous corresponding IF or ELSEIF conditions were found to be false. Execution continues to the corresponding ENDIF statement.

Example:

```
100 IF A < 0
110     I=-1
120     PRINT "A IS NEGATIVE"
130 ELSE
140     I=1
150     PRINT "A IS NOT NEGATIVE"
160 ENDIF
```

In this example, lines 110 through 120 are executed if A is less than zero, otherwise lines 140 through 150 are executed. In each case, execution then continues with the statement following line 160.

A.10 ELSEIF Statement

Syntax:

```
ELSEIF condition
```

Operation:

This statement must be used in conjunction with the IF and ENDIF statements. If all previous corresponding IF or ELSEIF conditions were found to be false, the specified "condition" is evaluated. If it is found to be true, execution continues with the next statement and proceeds until a corresponding ELSEIF, ELSE or ENDIF is encountered. Control then transfers to the first statement following the corresponding ENDIF. If the condition is found to be false, control is transferred to the next corresponding ELSEIF, ELSE or ENDIF.

Example:

```
100 IF A<0
110     I=-1
120     PRINT "A IS NEGATIVE"
130 ELSEIF A>0
140     I=1
150     PRINT "A IS POSITIVE"
160 ELSE
170     I=0
180     PRINT "A IS ZERO"
190 ENDIF
```

In this example, lines 110 through 120 are executed if A is negative; lines 140 through 150 are executed if A is positive; otherwise lines 170 through 180 are executed. In each case, execution continues with the statement following line 190.

A.11 END Statement

Syntax:

```
END
```

Operation:

This statement terminates the execution of a program. The CONT command can be used to resume execution at the statement following the END statement.

Example:

```
10 END
```

A.12 ENDIF Statement

Syntax:

```
ENDIF
```

Operation:

This statement specifies the end of a block of statements whose execution depends on some condition or conditions. The beginning of the block and the first condition are specified by the IF statement.

A.13 ENDLOOP Statement

Syntax:

```
ENDLOOP
```

Operation:

This statement specifies the end of a block of statements to be executed repeatedly. The beginning of the block is specified by the LOOP or WHILE statements.

Example:

```
10  I=1
20  WHILE I<3
30    PRINT I, SQR(I)
40    I=I+1
50  ENDLOOP
```

In this example statements 30 and 40 are repeated until I becomes greater than or equal to 3.

A.14 ENDPROC Statement

Syntax:

```
ENDPROC
```

Operation:

This statement specifies the end of a procedure or block of statements executed when an appropriate CALL statement is encountered. The beginning of the block of statements is defined by the PROC statement. When the ENDPROC statement is executed, control transfers to the statement following the CALL statement which invoked the procedure.

A.15 FOR Statement

Syntax:

```
FOR Index = Start TO Stop STEP Incr
```

Operation:

This statement defines the beginning of a block of statements to be executed repeatedly and the end of this block of statements is specified by the NEXT statement. "Index" must be a numeric variable and "Start", "Stop" and "Incr" must be numeric expressions. When the FOR statement is encountered during execution, "Index" is set to the value of "Start". Then the statements between the FOR and NEXT statements are executed. When the NEXT statement is encountered, the value of "Incr" is *added* to "Index". If "Incr" is positive and "Index" is greater than "Stop", or, if "Incr" is negative and "Index" is less than "Stop", execution continues with the statement following the NEXT statement. Otherwise, the statements between the FOR and NEXT statements are repeated. If STEP "Incr" is omitted, "Incr" is assumed to be 1.

Example:

```
10  FOR I = 0 TO 4 STEP 2
20    PRINT I, I*I
30  NEXT I
```

This example lists the even numbers from 0 through 4 and their squares.

A.16 GET Statement

Syntax:

```
GET variable
```

Operation:

The GET statement causes a single character to be read from the keyboard. "Variable" can be a numeric or string variable. If no character is available when this statement is executed, "variable" is still given a value. It is set to "" if it is a string variable, or 0 if it is a numeric variable.

Example:

```
10  LOOP
20    GET A$
30    IF A$ <> ""
40      PRINT A$
50    ENDIF
60  ENDLOOP
```

This example prints each key, as it is struck, on a separate line.

A.17 GET# Statement

Syntax:

```
GET#n, Variable
```

Operation:

This statement causes a single character to be read from the specified logical file n which logical file must have been opened with the OPEN statement. "Variable" can be a numeric or string variable. If no character is available when this statement is executed, "variable" is still given a value. It is set to "" if it is a string variable or 0 if it is a numeric variable.

Example:

```
10  OPEN 1,1,0, "DATAFILE"
20  GET#1,A$
30  PRINT ASC(A$)
40  IF A <>"" GOTO 20
```

This example lists characters from the tape file "DATAFILE" on the screen.

A.18 GOSUB Statement

Syntax:

```
GOSUB statement-number
```

Operation:

Execution of this statement causes control to transfer to the statement with the specified "statement-number". Execution continues until a RETURN statement is encountered at which time control returns to the statement following the GOSUB statement.

Example:

```
10  I=4
20  GOSUB 100
30  I=8
40  GOSUB 100
50  END
60  :
100 K=1
110 FOR J=1 to I
120   K=K*J
130 NEXT J
140 PRINT K
150 RETURN
```

This example computes and prints factorial 4 and factorial 8 by using the GOSUB statement to call the routine in lines 100 through 150.

A.19 GO TO Statement

Syntax:

```
GO TO statement-number
```

Operation:

This statement transfers control to the statement with the specified "statement-number".

Example:

```
10  I=1
20  PRINT I, I*I
30  I=I+1
40  GOTO 20
```

This example prints numbers and their squares on the screen indefinitely until terminated by the STOP key.

A.20 IF Statement

Syntax:

```
IF condition
```

Operation:

This statement specifies the beginning of a block of statements whose execution depends on some condition or conditions. The end of the block is specifies by the ENDIF statement. When executed, the "condition" is evaluated. If it is found to be true, execution continues with the next statement, and proceeds until a corresponding ELSEIF, ELSE or ENDIF is encountered. Control then transfers to the first statement following the corresponding ENDIF. If the condition is found to be false, control is transferred to the first corresponding ELSEIF, ELSE or ENDIF.

Example:

```
10 IF A>0
20   B=SQR(A)
30   PRINT A,B
40 ENDIF
```

In this example, the A and its square root are printed if A is non-negative.

A.21 IF-GO Statement

Syntax:

```
IF condition GOTO statement-number

IF condition THEN statement-number
```

Operation:

This statement causes the specified "condition" to be evaluated. If it is true, control is transferred to the specified "statement-number". Otherwise control continues with the next statement.

Example:

```
10  I=1
20  PRINT I, I*I
30  I=I+1
40  IF I<=10 GOTO 20
```

This example prints the numbers from 1 through 10 and their squares on the screen.

A.22 IF-THEN Statement

Syntax:

```
IF condition THEN statement
```

Operation:

This statement causes the specified "condition" to be evaluated. If it is true, the specified "statement" is executed, otherwise it is not.

Example:

```
10  INPUT X
30  IF X<0 THEN PRINT "X IS NEGATIVE"
```

This example reads a number from the keyboard and prints a message if it is negative.

A.23 IF-THEN-QUIT Statement

Syntax:

```
IF condition THEN QUIT
```

Operation:

This statement must occur within the block of statements defined by LOOP and ENDLOOP, WHILE and ENDLOOP, LOOP and UNTIL, WHILE and UNTIL, or IF and ENDIF. When encountered, the specified condition is evaluated. If it is found to be true, control is transferred to the statement following the block of statements; that is, the statement following the ENDLOOP, UNTIL or ENDIF.

Example:

```
100 LOOP
110     INPUT "ENTER NUMBER";N$
120     IF N$ = "STOP" THEN QUIT
130     SUM = SUM + VAL(N$)
140 ENDLOOP
```

In this example, lines 110 through 130 will be repeatedly executed until N$ becomes equal to the string "STOP". The statement in line 120 will detect this condition and control will be transferred to the statement following line 140.

A.24 INPUT Statement

Syntax:

```
INPUT variable,...
or
INPUT "prompt string"; variable,...
```

Operation:

These statements cause data to be read from the keyboard, converting the data to numbers or strings as specified by the type of the "variable"s. An optional prompt may be specified as "prompt string" followed by a semi-colon (;). The input data items must be separated by commas and must match the variables in both number and type. If the types don't match, the message "?REDO FROM START" appears and all the data items for that request must be re-entered. If too many data items are entered, the warning "?EXTRA IGNORED" is typed. If too few data items are entered, a "??" prompt is issued and additional data must be entered.

Example:

```
10  INPUT "ENTER THREE NUMBERS";A,B,C
20  PRINT A*A, B*B, C*C
```

This example issues a prompt, reads 3 numbers, and prints their squares. The user must type 3 numeric constants separated by commas.

A.25 INPUT# Statement

Syntax:

```
INPUT#n,variable,...
```

Operation:

This statement causes data to be transferred from the specified logical file, converting the data to numbers or strings as specified by the type of the "variable"s. The logical file must have been opened with the OPEN statement. Input data items in the file must be separated by commas and must match the variables in type, otherwise the message "?BAD DATA ERROR" is displayed.

A.26 LOOP Statement

Syntax:

```
LOOP
```

Operation:

This statement specifies the beginning of a block of statements to be executed repeatedly. The end of the block is specified by the ENDLOOP or UNTIL statements.

Example:

```
10  LOOP
20    INPUT X
30    IF X<0 THEN QUIT
40    PRINT SQR(X)
50  ENDLOOP
```

In this example, statements 20 through 40 are repeated until X becomes negative.

A.27 NEXT Statement

Syntax:

```
NEXT index
```

Operation:

This statement defines the end of a block of statements that are to be executed repeatedly. The beginning of the block is specified by the FOR statement. A description of the function of this combination of statements is found with the FOR statement description. The variable "index" in the NEXT statement must have the same name as the variable "index" in the corresponding FOR statement. If "index" is omitted, it is assumed to be the same as "index" in the corresponding FOR statement.

A.28 ON-GOTO Statement

Syntax:

```
ON variable GOTO stnum1, stnum2,...
```

Operation:

This statement causes control to transfer to another statement, depending on the value of the numeric variable "variable". If "variable"=1, then control transfers to the statement numbered "stnum1"; if "variable"=2, then control transfers to the statement "stnum2"; etc. If "variable"=0 or is greater than the number of statement numbers specified, execution continues with the next statement. If "variable" is <0 or >255, the message "ILLEGAL QUANTITY" is printed.

Example:

```
10   INPUT I
20   ON I GOTO 400, 500, 600
30   ...
```

In this example, if I=2, control goes to statement 500. If I=4, control goes to statement 30.

A.29 ON-GOSUB Statement

Syntax:

```
ON variable GOSUB stnum1, stnum2, ...
```

Operation:

This statement causes control to transfer to another statement depending on the value of the number "variable". If "variable"=1, control goes to the statement numbered "stnum1". If "variable"=2, control goes to the statement numbered "stnum2". If "variable"=0 or is greater than the number of statement numbers specified, control goes to the next statement. If "variable" is <0 or >255, the message "ILLEGIAL QUANTITY" is printed. If control transfers to a specified statement number; execution continues until a RETURN is encountered, then control returns to the statement following the ON-GOSUB.

A.30 OPEN Statement

Syntax:

```
OPEN logical-file, 1, secondary-addr, "filename"
```

Operation:

This statement prepares a file on the tape cassette for processing. The "logical file" can be any number between 1 and 255 and is used in INPUT#, PRINT#, GET#, and CLOSE statements pertaining to this file. The "secondary address" can be 0, 1, or 2. A 0 means the file is to be processed with INPUT# statements; a 1 means it is to be processed with PRINT# statements. A 2 means processing will be with PRINT# statements and an "end-of-tape" mark is to be written when the CLOSE statement is encountered.

A.31 POKE Statement

This statement is used to interface with the computer at the machine-language level. It's use is beyond the scope of this text. Information on this statement can be found in the appropriate Commodore manuals.

A.32 PRINT Statement

Syntax:

```
PRINT expression,...
or
PRINT expression;...
```

Operation:

This statement displays the value of the expressions on the screen. If two expressions are separated by commas (,), the second is aligned to begin at the beginning of a 10-character "window". Items of 10 characters or longer will use multiple windows. If two expressions are separated by semi-colons (;), they are printed with no intervening spaces. However, numeric expressions are always printed with a sign character preceding (- or blank) and a blank character following. If a PRINT statement ends with a semi-colon (;), the cursor will be positioned immediately following the last character printed; otherwise it is positioned at the beginning of the next line.

Example:

```
10 PRINT "HELLO";A$
```

This prints "HELLO" immediately followed by the value of the string A$.

A.33 PRINT# Statement

Syntax:

```
PRINT#n,expression,...
or
PRINT#n,expression;...
```

Operation:

This statement transfers the value of the expressions to the specified logical file n. This logical file must have been opened with the OPEN statement. If two expressions are separated by commas (,), the second is aligned at the beginning of a 10-character window. Items of 10 characters or longer will use multiple windows. If two expressions are separated by semi-colons (;), no intervening spaces are created in the logical file. However,

numeric expressions are always printed sign character preceding (- or blank) and a blank character following. If the PRINT# statement does not end with a semi-colon (;), a "carriage-return" character (CHR$(13)) is transferred to the file after the last item.

Example:

```
10 PRINT#3,"HELLO ";N$
```

This causes the strings "HELLO " and N$ to be transferred to the file represented by logical file number 3. A carriage-return character is appended to the end.

A.34 PROC Statement

Syntax:

```
PROC procedure-name
```

Operation:

This statement specifies the beginning of a procedure or block of statements to be executed when an appropriate CALL statement is encountered in the program. This CALL statement will reference the "procedure-name" defined in the PROC statement and the end of the block of statements is specified by the ENDPROC statement. The procedure name can be composed of an arbitrary number of characters.

Example:

```
100 PROC CHAR-COUNTER
110 SUM=0
120 I=LEN(S$)
130 WHILE I>0
140     IF MID$(S$,I,1)=C$
150       SUM=SUM +1
160     ENDIF
170 ENDLOOP
180 ENDPROC
```

This example is a procedure which will compute the number of occurrances of the character C$ in the string S$. The result is SUM. The procedure would be executed when the statement

```
CALL CHAR-COUNTER
```

was encountered in the program.

A.35 READ Statement

Syntax:

```
READ variable,...
```

Operation:

This causes data to be moved from DATA statements into the specified "variable"s. Data items are moved into variables in the order found on the DATA statement. When the data on one data statement is exhausted, data items are retrieved from the next one. If too few data items are found, the message "?OUT OF DATA" is given. The data items must match the type of the variable into which they are being moved.

Example:

```
READ I, A$, B$
```

This example would read a numeric constant and two string constants from DATA statements, assigning them to I, A$ and B$ respectively.

A.36 REM Statement

Syntax:

```
REM any characters
```

Operation:

The computer does not process the REM statement. It is included to allow documentation to be provided in the program.

Example:

```
10  REM THIS IS AN EXAMPLE.
20  REM WRITTEN BY J. WALTERS.
```

A.37 RESTORE Statement

Syntax:

```
RESTORE
```

Operation:

Execution of this statement allows the READ statements to re-read data specified in DATA statements. It resets the system to process the DATA statements from the beginning.

Example:

```
10  RESTORE
```

A.38 RETURN Statement

Syntax:

RETURN

Operation:

This statement must be used in a block of statements executed as the result of a GOSUB statement being encountered in the program. It causes control to transfer to the statement following the corresponding GOSUB statement. GOSUB-RETURN constructs can be nested to an unspecified level, depending on the characteristics of the active program.

A.39 STOP Statement

Syntax:

STOP

Operation:

This statement terminates the execution of a program. A message "BREAK IN LINE xxxx" is issued. The CONT command can be used to resume execution at the statement following the STOP statement.

Example:

```
400  STOP
```

A.40 SYS Statement

This statement is used to interface with the computer at the machine-language level. It's use is beyond the scope of this text. Information on this statement can be found in the appropriate Commodore manuals.

A.41 UNTIL Statement

Syntax:

```
UNTIL condition
```

Operation:

This statement specifies the end of a block of statements to be repeatedly executed. The beginning of the block of statements is specified by the LOOP or WHILE statements. The specified "condition" is evaluated *after* the enclosed block of statements is executed once. If the condition is found to be false, control returns to the beginning LOOP or WHILE. If the condition is found to be true, execution continues at the statement following the UNTIL.

Example:

```
10  I=1
20  LOOP
30    PRINT I, SQR(I)
40    I=I+1
50  UNTIL I>10
```

In this example, the numbers from 1 through 10 would be printed along with their square roots. Statements 30 and 40 are executed once and then repeatedly until I becomes greater than 10.

A.42 WAIT Statement

This statement is used to interface with the computer at the machine-language level. It's use is beyond the scope of this text. Information on this statement can be found in the appropriate Commodore manual.

A.43 WHILE Statement

Syntax:

```
WHILE condition
```

Operation:

This statement specifies the beginning of a block of statements to be repeatedly executed. The end of the block of statements is specified by the ENDLOOP or UNTIL statements. The specified "condition" is evaluated *before* the enclosed block of statements is executed. If the condition is found to be true, the enclosed block of statements is executed; then control returns to the WHILE where the condition is evaluated again. If the condition is found to be false, execution continues at the statement after the ENDLOOP or UNTIL.

Example:

```
10  INPUT I
20  WHILE I<3
30    PRINT
40    I=I+1
50  ENDLOOP
```

In this example, statements 30 and 40 will be executed only and as long as I remains less than 3. If I is set to 4 in statement 10, then statements 30 and 40 would not be executed at all.

Appendix B

System Commands

When the computer has typed READY, it is waiting for a command specifying the next action to be performed or for a statement to be added to the BASIC program. These system commands are described below.

B.1 CLR Command

Syntax:

```
CLR
```

Operation:

This resets the status of a BASIC program so that all variables are zero or null, and the system looks as if the program had just been loaded or typed in.

Example:

```
CLR
```

B.2 LIST Command

Syntax:

```
LIST stnum1-stnum2
```

Operation:

The LIST command displays the program in the workspace on the screen.

It displays lines numbered from "stnum1" through "stnum2". If "stnum2" is omitted, the program is listed from "stnum1" to the end. If "stnum1" is omitted, the program is listed from the beginning to "stnum2". If only LIST is specified, all lines in the program are listed.

Example:

```
LIST 100-
```

This command lists statement lines from line 100 to the end of the program.

B.3 LOAD Command

Syntax:

```
LOAD "program-name"
```

Operation:

This command loads a program with the name "program-name" from the tape cassette into the workspace. Any previous BASIC program in the computer will be erased. If "program-name" is not specified, the next file on the tape is loaded.

B.4 NEW Command

Syntax:

```
NEW
```

Operation:

This command clears the workspace of all programs, variables and data.

B.5 RUN Command

Syntax:

```
RUN statement-number
```

Operation:

This command causes the program in the workspace to begin execution at the specified "statement-number". If the statement number is omitted, execution begins at the lowest numbered statement. RUN does an implied CLR command before starting to execute the program.

B.6 SAVE Command

Syntax:

```
SAVE "program-name"
```

Operation:

This command saves the BASIC program in the workspace onto the tape cassette giving it the name "program-name". If "program-name" is not specified, the program is saved but is not given a name.

B.7 VERIFY Command

Syntax:

```
VERIFY "program-name"
```

Operation:

This command compares the program in the workspace with the program named "program-name" on the tape cassette. If "program-name" is omitted, the workspace program is compared with the "next" file on the tape. This command is often used to make sure the "SAVE" command was successful.

Appendix C

Intrinsic Functions

The following functions are "built-in" to the Commmodore BASIC System. Most of them take one or more arguments. In the description of each function, we will use the term S$ to indicate a "string argument" and N or M to indicate a "numeric argument". A string argument can be a string variable, constant, or expression; numeric argument can be a numeric variable, constant, or expression.

C.1 String Functions

C.1.1 ASC Function

Syntax:

```
ASC(S$)
```

Operation:

The ASC function takes one string argument. It returns the ASCII code equivalent of the first character in the specified string as an integer value. If S$ is null, the message "ILLEGAL QUANTITY" is displayed.

Example:

```
10  X=ASC("ABC")
```

This example sets X to the value 65 which is the ACSII code for "A".

C.1.2 CHR$ Function

Syntax:

```
CHR$(N)
```

Operation:

The CHR$ function takes one numeric argument. It returns a 1-character string which corresponds to the ASCII code specified as the numeric argument.

Example:

```
10  A$=CHR$(66)
```

This example returns the character "B" (ASCII code 66).

C.1.3 LEFT$ Function

Syntax:

```
LEFT$(S$,N)
```

Operation:

The LEFT$ function takes two arguments. The first is a string argument and the second is a numeric argument. The function returns a string composed of the left most N characters of the string S$. If N is not an integer, only the integer part is used.

Example:

```
10  A$ = LEFT("ABCDEF",2)
```

This example sets A$ to the string "AB".

C.1.4 LEN Function

Syntax:

```
LEN(S$)
```

Operation:

The LEN function takes one string argument. It returns the length of the specified string as an integer.

Example:

```
10  I=LEN("ABCD")
```

This example sets I to the value 4.

C.1.5 MID$ Function

Syntax:

```
MID$(S$,N,M)
```

Operation:

The MID$ function takes three arguments. The first is a string argument and the other two are numeric arguments. The function returns a string composed of M characters from the string S$, beginning with the N'th character of the string S$. If either N or M is not an integer, only the integer part is used.

Example:

```
10  A$=MID$("ABCDE",3,2)
```

This example sets A$ to the string "CD".

C.1.6 RIGHT$ Function

Syntax:

```
RIGHT$(S$,N)
```

Operation:

The RIGHT$ function takes two arguments. The first is a string argument and the second is a numeric argument. The function returns a string composed of the right-most N characters of the string S$. If N is not an integer, only the integer part is used.

Example:

```
10  A$ = RIGHT("ABCDEF",2)
```

This example sets A$ to the string "EF".

C.1.7 STR$ Function

Syntax:

```
STR$(N)
```

Operation:

The STR$ function takes one numeric argument. It returns a string which contains the characters that compose the number N.

Example:

```
10  A$ = STR$(327.03)
```

This example sets A$ to the string " 327.03".

C.1.8 VAL Function

Syntax:

```
VAL(S$)
```

Operation:

The VAL function takes one string argument. It converts the characters in the specified string to a numeric value. The string must contain the correct characters to compose a proper numeric constant. If they do not, the VAL function returns zero.

Example:

```
10  X = VAL("3.416")
```

This example converts the string "3.416" to the numeric value 3.416, and assigns it to X.

C.2 Arithmetic Functions

C.2.1 ABS Function

Syntax:

```
ABS(N)
```

Operations:

The ABS function takes one numeric argument. It returns a positive number equal to the absolute value of N.

Example:

```
10  X = ABS(-3)
```

This example sets X to 3.

C.2.2 ATN Function

Syntax:

```
ATN(N)
```

Operation:

The ATN function takes one numeric argument. It returns a number representing the arctangent of N, where N is expressed in radians.

Example:

```
10  A = ATN(.03)

A is set to .0299910049.
```

C.2.3 COS Function

Syntax:

```
COS(N)
```

Operation:

The COS function takes one numeric argument. It returns a number equal to the cosine of N, where N is expressed in radians.

Example:

```
10  X = COS(2)
```

This example sets X to -.416146836.

C.2.4 EXP Function

Syntax:

```
EXP(N)
```

Operation:

The EXP function takes one numeric argument. It returns a number equal to the constant "e" (approx. 2.71828183) raised to the power N.

Example:

```
10  X = EXP(3.1)
```

This example returns 22.1979513.

C.2.5 INT Function

Syntax:

```
INT(N)
```

Operation:

The INT function takes one numeric argument. It returns an integer equal to the largest integer which is less than or equal to N.

Example:

```
10  A = INT(2.1)
20  B = INT(-2.1)
```

In this example, A is set to 2 and B is set to -3.

C.2.6 LOG Function

Syntax:

```
LOG(N)
```

Operation:

The LOG function takes one numeric argument. It returns the natural logarithm of N. (i.e. log to the base "e" (2.71828183)). The numeric argument N must be greater than zero.

Example:

```
10  A = LOG(B)
```

C.2.7 π Function

Syntax:

```
π
```

Operation:

This function takes no arguments. It is used to provide the value of the numeric constant π (pi) which is 3.14159265. It can be used anywhere this number is required.

Example:

```
C = π *10
```

This example assigns C the value 31.4159265, which is the circumference of a circle with diameter 10.

C.2.8 RND Function

Syntax:

```
RND(N)
```

Operation:

The RND function takes one numeric argument. It returns a pseudo-random number between 0 and 1. Repeated calls to RND will generate a sequence of random numbers. (If the argument is a negative number, the same number is returned on each call to RND.)

Example:

```
10  PRINT 10 + RND(0)*8
```

This example generates a random number between 10 and 18.

C.2.9 SGN Function

Syntax:

```
SGN(N)
```

Operation:

The SGN function takes one numeric argument. If the argument is negative, SGN returns -1; if positive, it returns +1; and if zero it returns 0.

Example:

```
10  A = SGN(-32.1)
```

This example sets A to -1.

C.2.10 SIN Function

Syntax:

```
SIN(N)
```

Operation:

The SIN function takes one numeric argument. It returns a number equal to the sine of N, where N is expressed in radians.

Example:

```
10  B = SIN(3)
```

This example sets B to the sine of 3 which is .14112.

C.2.11 SQR Function

Syntax:

```
SQR(N)
```

Operation:

The SQR function takes one numeric argument. It returns a number equal to the square-root of N. The argument N must not be a negative number. If it is negative the message "?ILLEGAL QUANTITY" will be displayed. Accuracy is +5E-10.

Example:

```
10  PRINT SQR(14)
```

This prints the square root of 14 which is 3.74165739.

C.2.12 TAN Function

Syntax:

```
TAN(N)
```

Operation:

The TAN function takes one numeric argument. It returns a number equal to the tangent of N, where N is expressed in radians.

Example:

```
10  A = TAN(1)
```

This sets A to the tangent of 1 which is 1.55740772.

C.3 Special Purpose Functions

C.3.1 FRE Function

Syntax:

```
FRE(N)
```

Operation:

The FRE function takes a dummy numeric argument. It returns a number which is equal to the number of "bytes" of free space remaining in the computer for program lines and variables. Each character in a program line usually occupies one "byte" of storage; each numeric variable occupies 7 bytes of storage; each character in a string occupies one byte of storage. Each time the FRE function is used, the system will consolidate all unused bytes of storage in an operation called "garbage collection".

Example:

```
10 X=FRE(0)
```

This statement assigns a number to X equal to the number of bytes of storage available.

C.3.2 PEEK Function

This function is used to interface with the computer at the machine-language level. Its use is beyond the scope of this text. Information on this function can be found in the appropriate Commodore manuals.

C.3.3 POS Function

Syntax:

```
POS(N)
```

Operation:

This function takes a dummy numeric argument. It returns a number equal to the column number of the cursor at that time. The cursor position is zero after the return key has been pressed. (The "HOME" and "CLR" keys do not affect the POS function).

Example:

```
10 PRINT POS(0)
```

This statement will print a number equal to the current column number of the cursor.

C.3.4 SPC Function

Syntax:

```
SPC(N)
```

Operation:

The SPC function is used to skip over a specified number of positions on the screen. Characters already on the screen are not modified. The

numeric operand N defines the number of columns to be skipped. If N is less than 0 or greater than 255, the message "?ILLEGAL QUANTITY" is displayed.

Example:

```
10 PRINT "A"; SPC(10); "B"
```

This statement will skip 10 positions between printing the character "A" and the character "B".

C.3.5 ST Function

Syntax:

```
ST
```

Operation:

This function takes no arguments. It returns a number representing the status of the most recent input-output operation. Since the meaning of the numbers is different for different devices, consult the manual describing specific devices for details. The cassette unit returns the following status values for the specified error conditions.

0	- no errors.
4	- a short block was read.
8	- a long block was read.
16	- unrecoverable error or verify mismatch.
32	- checksum error
64	- end of file detected.
-128	- end of Tape detected.

C.3.6 TAB Function

Syntax:

```
TAB(N)
```

Operation:

The TAB function is used to position the cursor at a specific column on the screen. Characters already on the screen are not modified. The numeric operand N defines the column number. Columns are numbered from 0 and wrap around to succeeding lines on the screen. If the cursor is already past the column specified by N, no spacing is performed. The message "?ILLEGAL QUANTITY" will be displayed if the number N is less than 0 or greater than 255.

Example:

```
10 PRINT "A"; TAB(20); "B"
```

This statement prints the character "A" in column zero and the character "B" in column 20.

C.3.7 TI Function

Syntax:

```
TI
```

Operation:

This function takes no arguments and is used to provide the value of the real-time clock in the Commodore BASIC System. It always contains a number which represents the time since the machine was turned on in units of 1/60ths of a second.

Example:

```
10 PRINT TI/60
```

This example prints the number of seconds since the machine was turned on.

C.3.8 TI$ Function

Syntax:

```
TI$
```

Operation:

This function is used to maintain a 24-hour clock in a BASIC program. It contains a string of 6 characters, the first two representing hours, the next two representing minutes, and the last two representing seconds. The clock can be "set" by assigning the appropriate string to TI$. Subsequently it can be used by referencing TI$.

Example:

```
10 TI$ = "102417"
```

This statement sets the 24-hour clock to 10 hours, 24 minutes and 17 seconds past midnight. hours and 3 minutes later, the statement

```
20 PRINT TI$
```

was executed, the string "152717" would be displayed.

C.3.9 USR Function

This function is used to interface with the computer at the machine-language level. Its use is beyond the scope of this text. Information on this function can be found in the appropriate Commodore manuals.

Appendix D

Reserved Words

Various words in the BASIC language are used for special purposes. These words cannot be used anywhere in a program (except comments) for any other than their special purpose. Specifically, they cannot be used as variable names or as parts of variable names. Illegal use of these reserved words usually causes the message "?SYNTAX ERROR" to occur. For example, the variable name MORE cannot be used because it contains the reserved word OR imbedded in it.

```
ABS
AND
ATN
ASC

CALL
CHR$
CLOSE
CLR
CMD
CONT
COS

DATA
DEF
DIM

ELSE
ELSEIF
END
ENDIF
ENDLOOP
ENDPROC
EXP

FN
FOR
FRE

GET
GET#
GO
GOSUB
GOTO

IF
INPUT
INPUT#
INT

LEFT$
LEN
LET
LIST
LOAD
LOG
LOOP

MID$

NEW
NEXT
NOT

ON
OPEN
OR

PEEK
POKE
POS
PRINT
PRINT#
PROC

QUIT

READ
REM
RESTORE
RETURN
RIGHT$
RND
RUN

SAVE
SGN
SIN
SPC(
SQR
STEP
STOP
STR$
SYS

TAB(
TAN
THEN
TO

USR
UNTIL

VAL
VERIFY

WAIT
WHILE
```

Appendix E

Full-Screen Editing

The Commodore microcomputers have a *full-screen editor* which allows the user to modify the text appearing on the screen. It is possible to change or delete characters and also to insert new characters. The next few paragraphs describe how these functions work.

E.1 The Keyboard and The Screen

Although the keyboard is not connected to the screen, every time we strike a key, the character corresponding to that key will appear on the screen. Where does this character appear? There is a bright flashing square on the screen called a *cursor* and when a key is struck the cursor is replaced by the character and the cursor moves one position to the right.

If one wishes to place characters at different points on the screen, the cursor can be moved to the appropriate position by using the two *cursor control keys* on the keyboard. These keys are marked with the letters CRSR. The cursor can be moved either to the left or right, or up or down by repeatedly depressing one of the keys. The left or up motions can be obtained by depressing the SHIFT key at the same time as the appropriate CRSR key is depressed. The reader might wish to experiment with these keys.

If the cursor is at the extreme right of the screen, then moving the cursor one more position to the right will force the cursor to appear on the extreme left and one line lower. Moving the cursor to the extreme left and then moving one more position to the left will force the cursor to move to the extreme right and one line higher. This operation is called *wrap-around.*

Moving the cursor to the top or bottom of the screen will cause different effects. The cursor can not be moved beyond the top of the screen with the CRSR key. If the cursor is placed at the bottom of the screen and then the CRSR key is used to move the cursor lower, everything on the screen shifts up one line. This means the top line on the screen disappears and a blank line appears at the bottom. This particular effect is known as *scrolling.*

The *CLEAR/HOME key* is also used to manipulate the cursor. If you depress this key the cursor moves to the upper left corner of the screen. If you depress this key and SHIFT at the same time then the screen is cleared and cursor moves to the upper left corner.

The following sections describe how to use the cursor to edit a line.

E.2 Changing Characters

Move the cursor to the character to be changed, type the new character and depress RETURN.

In the statement

```
100  X = X + 3
```

the 3 can be changed to a 2 by moving the cursor onto the 3, typing 2 and depressing RETURN.

E.3 Deleting Characters

Move the cursor to the character which is one character to the right of the one to be deleted. Depress the Delete key once for each character to be deleted and then depress RETURN.

In the statement

```
100 X = Y + 73.5
```

the 73.5 can be changed to .5 by moving the cursor to the period (.) and depressing the Delete key twice followed by RETURN.

E.4 Inserting Characters

Move the cursor to the character which is to appear to the right of the new chararacter and depress the Insert key. A blank space will appear and the character to be inserted can now be typed. The insert key must be depressed for each new character to be inserted. After all characters have been inserted in a line, depress the RETURN key.

In the statement

```
100  X = Y + .5
```

the .5 can be changed to 73.5 by moving the cursor to the period (.), depressing Insert twice, then typing 7 followed by 3 followed by RETURN.

E.5 Replacing or Deleting an Entire Line

To replace or delete an entire line in BASIC, type the line number followed by the new text and then depress the RETURN key. If no text appears after the line number then the line is deleted.

To delete the statement

```
100  X = X + .3
```

type the statement number 100 followed by RETURN.

To replace the statement

```
100  X = X + .3
```

by

```
100  Y = Y - .5
```

type the statement number 100 followed by "Y = Y - .5" and a RETURN.

Notes:

(1) You must always depress the RETURN key after all changes have been made to a line; otherwise the line will not be modified.

(2) Editing on the screen may cause a program listing to appear strange. If you are not sure what you have done to a program then use LIST to obtain a new copy on the screen.

Appendix F

Messages in BASIC

The BASIC system displays a number of messages to inform the user about the type of activity which is occurring, and to indicate whether a previous operation caused an error.

F.1 Possible BASIC Error Messages and Meanings

A number of errors often occur in programs as they run. The BASIC system analyses a program as it is running and if an error is detected an error message is displayed on the screen. This appendix lists the error messages and describes some probable causes for the errors.

The error messages are mostly of the form

?error message in XXX

or

error message in XXX

where XXX is a line number in the BASIC program. A small number of the messages do not refer to a specific line and hence do not contain a line number. Once an error is detected by BASIC it is often not possible to continue execution of the program. You can usually determine whether the program has failed by examining the text accompanying each message in this Appendix. Variables within the statement in error and the program

retain their values so they may be examined to determine the cause of the error.

F.1.1 ??

The INPUT statement continues to function until acceptable data has been received. When not enough data has been typed in response to INPUT, a double question mark (??) is printed until enough data is received.

```
10 INPUT A,B,C
RUN
?1
??2
??3
READY.
```

F.1.2 BAD SUBSCRIPT

An attempt was made to reference a matrix element which is outside the dimensions of the matrix. This may happen by specifying the wrong number of dimensions or a subscript larger than specified in the original DIM statement.

```
DIM A(2,2)
A(1,1,1)=2
?BAD SUBSCRIPT ERROR
READY.
A(10,10)=2
?BAD SUBSCRIPT ERROR
READY.
```


F.1.3 CAN'T CONTINUE

Program execution cannot be resumed via a CONT command. There are four possible reasons: 1) no program exists. 2) a new line was just typed in. 3) the program has not recently been run. 4) an error just occurred.

```
10A$='HELLO'
CONT
'CAN'T CONTINUE ERROR'
READY.
```

F.1.4 DIVISION BY ZERO

Zero as a divisor would result in numeric overflow - thus it is not allowed. When this message appears, it is most expedient to list the statement and look for division operators.

```
?DIVISION BY ZERO ERROR IN 10
LIST 10
10A=B/C
?C
0
```

F.1.5 FORMULA TOO COMPLEX

This message is only pertinent to string expressions and indicates that BASIC does not have enough space to evaluate the whole string. This problem can be solved by breaking the string into two smaller strings in two statements and then combining these two strings in a third statement.

```
?FORMULA TOO COMPLEX
```

F.1.6 ILLEGAL DIRECT

INPUT and DEF cannot be used in direct commands; this error indicates that one of these commands has been tried in that mode.

```
?ILLEGAL DIRECT ERROR
```

F.1.7 ILLEGAL QUANTITY

This error occurs when a function is accessed with a parameter out of range. This error may be caused by:

(1) A matrix subscript not in the range 0 to 32767

```
X(-1)=Y
?ILLEGAL QUANTITY ERROR
```

(2) LOG (negative or zero argument)

(3) SQR (negative argument)

(4) Call of USR before machine language subroutine has been patched in.

(5) Use of string functions MID$, LEFT$, RIGHT$, with length parameters not in the range 1 to 255.

(6) Index on ...GOTO out of range.

(7) Addresses specified for PEEK, POKE, WAIT and SYS not in the range 0 to 65535.

(8) Byte parameters of WAIT, POKE, TAB and SPC not in the range 0 to 255.

```
POKE 32768,1000
?ILLEGAL QUANTITY ERROR
READY.
```

F.1.8 INVALID IF

An IF statement is constructed incorrectly

```
IF A=B CALL XYZ
INVALID IF ERROR IN XXX
```


F.1.9 NEXT WITHOUT FOR

Either a NEXT is improperly nested or the variable in a NEXT statement corresponds to no previously executed FOR statement.

```
10 FOR I=1 TO 10
20 NEXT
30 NEXT
?NEXT WITHOUT FOR ERROR
READY.

10 FOR I=1 TO 10
20 NEXT J
?NEXT WITHOUT FOR ERROR
READY.
```

F.1.10 OUT OF DATA

A READ statement was executed but all of the data statements in the program have been read. The program tried to read too much data, or insufficient data was included in the program. Depressing carriage return when the cursor is on a READY message sometimes yields this error because the message is interpreted as READ Y.

```
READY.
?OUT OF DATA ERROR
READY.
```

F.1.11 OUT OF MEMORY

This message may appear while entering or editing a program as the text completely fills memory. At run time, assignment and creation of variables may also fill all variable memory. Array declarations consume large areas of memory even though a program may be rather short. The maximum number of FOR loops and simultaneous GOSUBs allowed are dependent on each other and this maximum capacity may be exceeded. To determine the type of memory error, type ?FRE(0). If there are a large number of bytes available, it is most likely a FOR-NEXT or GOSUB problem.

```
10 GOSUB 10
RUN
?OUT OF MEMORY ERROR IN 10
READY.
?FRE(0)
7156
(This is a FOR-NEXT or GOSUB problem.)
```

F.1.12 OVERFLOW

Numbers resulting from computations or input that are larger than 1.70141183 E + 38 cannot be represented in BASIC's number format. Underflow is not a detectable error but numbers less than 2.93873588 E-39 are indistinguishable from zero.

```
?1E40
?OVERFLOW ERROR
READY.
```


F.1.13 REDIM'D ARRAY

After a matrix was dimensioned, another dimension statement for the same matrix was encountered. For example, an array variable is defined by default when it is first used, and later a DIM statement is encountered.

```
A(5)=6
DIM A(10,10)
?REDIM'D ARRAY ERROR
READY.
```

F.1.14 REDO FROM START

This error is not actually a fatal error but is a diagnostic printed when data supplied to the INPUT statement is alphabetic when a numeric quantity is required.

```
10 INPUT A
RUN
?ABC
?REDO FROM START
?
```

F.1.15 RETURN WITHOUT GOSUB

A RETURN statement was encountered without a previous GOSUB statement being executed.

```
CLR
RETURN
?RETURN WITHUT GOSUB ERROR
```

F.1.16 STRING TOO LONG

Attempt by use of the concatenation operator to create a string more than 255 characters long.

```
10 A$='A'
20 FOR I=1 TO 10:A$=A$:NEXT
30 A$=A$+A$
40 NEXT
?STRING TOO LONG ERROR
READY.
```

F.1.17 SYNTAX

BASIC cannot recognize the statement you have typed. This error is caused by such things as missing parentheses, illegal characters, incorrect punctuation, or a misspelled keyword.

```
RUIN
?SYNTAX ERROR
READY.
```

F.1.18 TYPE MISMATCH

The left-hand side of an assignment statement was a numeric variable and the right-hand side was a string, or vice versa; or a function which expected a string argument was given a numeric one, or vice versa.

```
A$=5
?TYPE MISMATCH ERROR
READY.
```

F.1.19 UNDEF'D STATEMENT

An attempt was made to GOTO, GOSUB or THEN to a statement which does not exist.

```
GOTO A
?UNDEF'D STATEMENT ERROR
READY.
```


F.1.20 UNDEF'D FUNCTION

Reference was made to a user defined function which had never been defined.

```
X=FNA(3)
?UNDEF'D FUNCTION ERROR
READY.
```

F.1.21 UNDEFINED PROC

The program attempts to call a procedure which has never been defined. An example follows.

```
100  CALL ABC
       .
       .
       .
    (end of program)
(The procedure was not defined)
UNDEFINED PROC ERROR IN XXX
```

F.1.22 UNMATCHED STRUCTURE

The structure starting with LOOP, WHILE or IF has no corresponding ENDLOOP or ENDIF statement. An example illustrates this error.

```
10  IF A<B
      .
      .
      .
    (end of program)
UNMATACHED STRUCTURE ERROR IN XXX.
```

F.2 Operating System Messages and Meanings

The BASIC system also monitors the reading of files from a cassette and displays messages to indicate errors that have occurred.

F.2.1 BAD DATA

Numeric data was expected but alphabetic data was received when obtaining input from a special device.

F.2.2 DEVICE NOT PRESENT

The requested device is not attached to the computer. The status indicator will have a value of 2 which corresponds to a time out. This error may happen on OPEN, CLOSE, CMD, INPUT#, GET#, or PRINT#.

```
OPEN 5,4,3 'FILE'
?DEVICE NOT PRESENT ERROR
READY.
```

F.2.3 FILE NOT FOUND

The named file specified in OPEN or LOAD was not found on the device specified. In the case of tape I/O an end of tape mark was encountered.

F.2.4 FILE NOT OPEN

The operating system must have device number and command information provided by the OPEN statement. If an attempt is made to read or write a file without having done this previously, then this message appears:

```
CLR
INPUT#10,A
?FILE NOT OPEN ERROR
READY.
```

F.2.5 FILE OPEN

This error indicates an attempt to redefine file parameter information by repeating an OPEN command on the same file twice.

```
OPEN 1,4,1
OPEN 1,4,1
?FILE OPEN ERROR
READY.
```

F.2.6 LOAD

Only occurs when loading a program from cassette tape. This means that there were more than 31 errors in the first tape block or that there were errors in exactly the same corresponding positions of both blocks.

F.2.7 NOT INPUT FILE

Tape files, once opened for writing, cannot be read without first closing the file, rewinding the tape and opening the file for INPUT. This message appears when an attempt is made to read an output file:

```
10 OPEN 1,1,1
20  INPUT #1,A
?NOT INPUT FILE ERROR
READY.
```

F.2.8 NOT OUTPUT FILE

Tape files cannot be read and updated in place. For example device 0 is the keyboard and a program cannot write to it.

```
10  OPEN 1,0
20  PRINT#1
?NOT OUTPUT FILE ERROR
READY.
```

F.2.9 VERIFY

The contents of memory and a specified file do not compare.

Appendix G

Input and Output to Disks and Printers

Chapter 2 introduced some basic concepts in manipulating programs. The commands SAVE, LOAD and VERIFY were used to store and retrieve program files on cassette tape. Chapter 8 showed how the statements OPEN, CLOSE, INPUT#, and PRINT# can be used to create and process data files on tape.

In the following sections, we will see how these facilities can also be used to manipulate program and data files on the Commodore disk and printer.

G.1 Simple Output to the Printer

The Commodore printer is connected to the computer through an external bus connector. A program can produce printed output by writing a file to the printer.

```
1000 REM EXAMPLE G.1
1010 :
1020 OPEN 1,4
1030 X=1
1040 WHILE X<=10
1050   PRINT#1,X,SQR(X)
1060   X=X+1
1070 ENDLOOP
1080 CLOSE 1
1090 STOP
```

In Example G.1 the OPEN statement designates the printer, device number 4, as logical file 1. Any lines PRINTed to this logical file will appear on the printer. Line 1050 will be executed 10 times and produce a table of the square roots of the numbers from 1 through 10. When the output is complete, the CLOSE statement is used to "disconnect" the printer from logical file 1.

G.2 Listing a Program on the Printer

It is often desirable to produce a copy of a program on paper for reference or documentation. However, the LIST command displays the program statements on the primary output device, namely the screen. It is possible, however, to assign the printer as the primary output device, and then list the program there. To do this, type the following two statements at the keyboard.

```
OPEN 1,4
CMD 1
```

These statements prepare the printer, device 4, as the "primary output device". Now, typing commands at the keyboard causes the response to be printed on the printer. Simply type

```
LIST
```

and the statements of the current program will be listed on the printer.

When this operation is complete and the cursor reappears, the printer must be "disconnected" as the primary output device by typing the two commands:

```
PRINT#1
CLOSE1
```

Everything should now be back to normal and the screen should operate as before.

G.3 Preparing to Use the Disk

The Commodore disks can be used to store and retrieve both program and data files. Each diskette is first prepared for use by a process called "NEWing". This erases all information from the diskette and prepares it to contain any new data or programs.

The disk can contain two diskettes at one time. These are inserted in drive 0 or drive 1, indicated on the front of the unit. The command:

```
OPEN 1,8,15,"NO:TEST,99"
```

will cause the diskette in drive 0 of the disk (device 8) to be "NEWed" and given a name of TEST with id of 99. During this operation, the light on the disk associated with drive 0 will be on. When the light goes out, the process is complete and the command:

```
CLOSE 1
```

should be issued. If an error occurs during this process, the "error light" in the centre of the disk unit will be on. Consult the Commodore manual for error diagnostic procedures.

Once a diskette has been "NEWed", this procedure need not be repeated unless all the information is to be erased. However, each subsequent time a diskette is to be used, it must be "initialized" using the command

```
OPEN 1,8,15,"IO"
```

This "initializes" the diskette in drive 0 for use by the computer. The command

```
CLOSE 1
```

should then be issued. Programs and data files can now be stored and retrieved from the diskette.

Notes:

(1) In the OPEN commands above, the 1 is any logical file number. The 8 refers to the disk unit which is device 8. The number 15 in the third parameter denotes the "command channel".

(2) The INITIALIZE operation must be done whenever a diskette is inserted into the drive.

G.4 Storing Programs on Disk

Once a diskette has been "NEWed" and "Initialized", programs can be stored on it in the form of program files. Simply type the command

```
SAVE "0:PROGRAM",8
```

and a file named PROGRAM will be created on the diskette in drive 0 containing the program currently in the workspace. The "0:" at the front of the name designates the drive number. The ",8" specifies disk drive 8.

It is possible, although infrequent, that the SAVE operation may terminate successfully, (i.e. the error light is off) but the program will be stored incorrectly. The VERIFY command, described in Chapter 2, can be used to validate the SAVE operation. Type

```
VERIFY "0:PROGRAM",8
```

and the contents of the specified file are compared to the program in the workspace.

At some later time, you may want to copy the program from disk back into the workspace. Make sure the diskette is inserted correctly and initialized. Then type

```
LOAD "0:PROGRAM",8
```

this causes a copy of the program to be loaded into the workspace.

The above commands can be used to store many programs on a single diskette. They can be selectively "LOADed" by specifying the appropriate name. It is possible to load a program specifying only the first few characters of the name. Typing

```
LOAD "0:PR*",8
```

will load the first program encountered whose name begins with the character sequence "PR".

NOTE: Files created with the SAVE command are said to be files of type PRG.

G.5 Data Files on Disk

One of the most powerful facilities provided with a disk is the ability for the program to process files of data. Information can be stored and analyzed and new data created.

```
100 REM EXAMPLE G.3
105 :
130 OPEN 6,8,5,"0:TELEPHON,SEQ,WRITE"
135 :
150 LOOP
155   PRINT "ENTER NAME AND PHONE"
160   INPUT NAME$, NUMBER$
170   IF NAME$="ZZZZ" THEN QUIT
180   PRINT#6,NAME$;",";NUMBER$
190 ENDLOOP
195 :
200 PRINT#6,"ZZZZ,999-9999"
220 CLOSE 6
225 :
230 STOP
```

Example G.3 is the same as Example 8.2 in that it creates a file of names and telephone numbers. The OPEN statement in line 30, however, specifies that the file is to be created on the diskette in drive 0 of device 8. The third parameter specifies that buffer number 5 in the disk is to be used for temporary storage. This is referred to as the "secondary address" and can be any number from 2 through 14. If more than one file is open at the same time on the same device (i.e. number 8) the secondary address must be unique for each file that is open.

The fourth parameter, enclosed in quotes, denotes the drive number, name, type and access of the file. The parameter:

```
"0:TELEPHON,SEQ,WRITE"
```

specifies a file named TELEPHON on drive 0 is to be a sequential file (SEQ) and is to be written (WRITE) using PRINT# statements.

Data records are written to the file in line 180 and line 200 much the same way as they were written to cassette in example 8.2. When all desired records have been written, the CLOSE command is issued.

NOTE: When using disks connected via the IEEE port, the PRINT statements must appear as follows:

```
180 PRINT#6,NAME$;",";NUMBER$;CHR$(13);
200 PRINT#6,"ZZZZ,999-9999";CHR$(13);
```

To recall the information stored in a sequential disk file, another program must be written.

```
100 REM EXAMPLE G.4
105 :
130 OPEN 6,8,4,"0:TELEPHON,SEQ,READ"
135 :
150 LOOP
160   INPUT#6,NAME$,NUMBER$
170   IF NAME$="ZZZZ" THEN QUIT
180   PRINT NAME$,NUMBER$
190 ENDLOOP
220 CLOSE 6
225 :
230 STOP
```

Example G.4 reads the names and telephone numbers from the file TELEPHON and prints them on the screen. The OPEN statement assigns logical file number 6 to the file TELEPHON on the diskette in drive 0 of device number 8. The secondary address of 4 specifies that disk buffer number 4 is to be used for this file. The file is sequential (SEQ) and is to be read (READ) using INPUT# statements.

Records are read from the file using the INPUT# statement in the normal manner. When all records are read, the file is CLOSEd in line 220.

G.6 Removing a File from Disk

A file can be removed from the disk by sending it a special command via the "command channel". This is done by using the secondary address 15. For example, the statements


```
OPEN 1,8,15,"S0:TELEPHON"
CLOSE 1
```

are used to send the command

```
"S0:TELEPHON"
```

to the command channel (15) of disk drive 8. The parameter specifies a file named TELEPHON on drive 0 is to be scratched (S).

When the disk detects an error, a red light on the disk panel is turned on. The nature of the error can be discovered by reading certain information from the disk's "command channel". For example, the statements

```
OPEN 1,8,15
INPUT#1, E$,M$,T$,S$
PRINT E$, M$, T$, S$,
```

would cause 4 strings to be read from the command channel (15) of the disk and displayed on the screen. The nature of the error is contained in the strings as follows:

E$ - the number of the error message

M$ - the text of the error message

T$ - 00 or the track number where the error occurred

S$ - 00 or the sector number where the error occurred

If E$ is 00, then no error was detected and everything can proceed as normal. Other values indicate errors which are described in the appropriate disk manual.

G.7 The Disk Directory

Each diskette can contain a number of separate program and data files. Consequently, each diskette contains a "directory" to enable the system to keep track of these files. It is possible to display this directory by first LOADing it into the workspace and then LISTing it on the screen. Type

```
LOAD "$0",8
```

to load the directory of the diskette in drive 0 of disk device number 8. (LOAD "$1",8 would load the directory for drive 1.) Then type

```
LIST
```

to display this directory on the screen. It should look something like this:

```
0 "TEST              " 99
1    "PROGRAM"           PRG
3    "FILE A"            SEQ
3    "FILE B"            SEQ
57   "SAILOR"            PRG
606 BLOCKS FREE.
```

By referring to the first line, we can see that this directory describes the contents of the diskette in drive 0. It is called TEST with an id code of 99 and currently contains 4 files. These are called "PROGRAM", "FILE A", "FILE B", and "SAILOR" occupying 1,3,3, and 57 "disk blocks" respectively. A disk block consists of 256 characters or bytes of data. Two of the files, namely "PROGRAM" and "SAILOR", contain programs (type PRG); the others are sequential data files (type SEQ). This diskette can contain an additional 606 disk blocks of programs or data.

NOTE: Program and data files can be given names up to 16 characters in length. If longer names are specified, only the first 16 characters are used.

A "USER FRIENDLY" COMPUTER

The new VIC computer is designed to be the most user friendly computer on the market...friendly in price, friendly in size, friendly to use and expand.

With the VIC, Commodore is providing a computer system which helps almost anyone get involved in computing quickly and easily...with enough built-in expansion features to let the system "grow" with the user as his knowledge and requirements become more sophisticated.

VIC owners who wish to learn more about computing should ask their Commodore dealer about these other self-teaching and reference materials:

- **VIC LEARNING SERIES...** a library of self-teaching books and tapes/cartridges which help you learn about computing and other subjects. Volume I in the VIC Learning Series is called "Introduction to Computing...On the VIC". Volume II is called "Introduction to BASIC Programming". Subsequent titles will include Animation, Sound and Music, and more.

- **VIC PROGRAMMER'S REFERENCE GUIDE...**a comprehensive guide to the VIC20 Personal Computer, including important information for new and experienced programmers alike.

- **VIC-PROGRAM TAPES, CARTRIDGES AND DISKS...**a growing library of recreational, educational and home utility programs which let you use the VIC to solve problems, develop learning skills, and play exciting television games. These easy-to-use programs require no previous computer experience.

commodore COMPUTER

1200 Wilson Drive,
West Chester,
Pennsylvania, 19380
U.S.A.

3370 Pharmacy Ave.,
Agincourt,
Ontario, Canada
M1W 2K4

Printed in Canada