# SuperPET

# Waterloo microBASIC ©

# Waterloo microBasic

# Tutorial and Reference Manual

J. Wesley Graham

K. Ian McPhee

**Disclaimer**

Waterloo Computing Systems Limited makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall Waterloo Computing Systems Limited, its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claim for lost profits, fees or expenses of any nature or kind.

*Preface*

Waterloo microBASIC is an interactive BASIC language interpreter which provides simple, comprehensive facilities for entering, running, debugging and editing programs. The programming language supports many important extensions beyond standard BASIC such as structured programming control, long names for variables and other program entities, procedures callable with parameters, multi-line functions, sequential and relative file capabilities, integer arithmetic, MAT statements for matrix operations, powerful character-string manipulation, and a broad set of intrinsic functions. Waterloo microBASIC includes ANS BASIC as defined in the 1978 X3.60 standard.

This manual describes the Waterloo microBASIC processor in a general manner applicable to use with various computer systems. Specific examples, in some cases, illustrate its use with the Commodore SuperPET microcomputer. System-specific information, in general, can be found in the Waterloo System Overview manual which applies to the system in use. This manual is organized into four parts. The first part introduces the reader to the general characteristics of the system and contains a collection of annotated examples illustrating features of the programming language. The next two parts comprise a comprehensive reference manual describing the Waterloo microBASIC command and programming languages, respectively. The fourth part consists of appendices summarizing the command and programming languages and describing use of files with Waterloo microBASIC.

*Acknowledgement*

All members of the Computer Systems Group at the University of Waterloo have made a significant contribution to the design of the Waterloo microBASIC interpreter. The design is based upon ideas evolved and proven over the past decade in other software projects in which the group has been involved. The actual design and programming of the interpreter was performed by Fred Crigger, Carl Durance and Ian McPhee. Charlotte Ross was responsible for production of the manual.

J.W. Graham
K.I. McPhee

June 1981

Table of Contents

# Tutorial

# Command Language Reference Guide

# Programming Language Reference Guide

# Appendices

# Waterloo microBASIC

# Tutorial

# Chapter 1

# A Flavour of Waterloo microBASIC

## 1.1 Introduction

The following tutorial contains a sequence of examples meant to introduce the reader to the "flavour" of the Waterloo microBASIC programming language. The examples are preceded by a very brief introduction to some Waterloo microBASIC commands to enable the reader to try the example programs on the computer.

This tutorial does not present a complete or rigorous treatment of any topic, as this detailed information is available in the reference sections of this manual. This tutorial could be useful in the following situations:

- Someone already familiar with BASIC can determine some of the major differences between Waterloo microBASIC and the dialect already known.
- Teachers may find the examples useful as a progressive introduction of the material to their students.
- People who already know some other language can get an appreciation for Waterloo microBASIC before reading the reference sections.

- Complete novices could run the various programs, and possibly learn some of the material by exploring the various features in conjunction with the reference material.

## 1.2 Command Language Introduction

If you have a flexible disk containing copies of these BASIC examples, you can easily try each example without having to type it. If this flexible disk is placed in drive 0 of your disk drive, each program can be copied into the workspace and executed with a single command. For example, if the first BASIC example is in a file named "BEX1", the command

```
RUN "BEX1"
```

will copy it into the workspace and execute it.

### 1.2.1 Sign-on Procedure

When the computer is turned on, you are asked to select a processor. Type the letter b for BASIC.

### 1.2.2 Typing in a Program

When you have signed on, you are alloted a *workspace* to hold your program. This space is empty and the system is READY for typing in the first line of the program. If you type

```
10 X=1
```

and press the RETURN key, the basic statement X=1 is entered into the workspace as line 10. If you then type

```
5 Y=1
```

and press RETURN the computer automatically places this statement in *line 5* of the workspace, ahead of the previous line.

Example 1 is typed in as follows:

```
30 x = 12
40 y = x*x
50 print x,y
60 stop
```

### 1.2.3 Listing Your Workspace

At any time, the current contents of the workspace can be seen on the screen by typing

```
LIST
```

If you type LIST 20 then only line 20 is displayed, whereas if you type LIST 20-100 all lines from 20 to 100 inclusive are displayed. The LIST command and all other commands can be typed in upper or lower case (e.g., LIST or list). Commands are shown in uppercase here.

### 1.2.4 Running a Program

When a program has been entered into the workspace it is put into operation by typing RUN. The program is still there even after it has been run and can be started again by typing RUN.

### 1.2.5 Saving a Program

If you wish to store your program, type

```
SAVE 'CHARLIE'
```

and the whole workspace will be saved in your file on your disk with the name 'CHARLIE'.

### 1.2.6 Retrieving a Program

A previously saved program can be read into your workspace by typing

```
OLD 'CHARLIE'
```

This causes the previous contents of the workspace to be erased and the file named 'CHARLIE' to be read in.

### 1.2.7 Changing a Program

(i) *Adding Lines*: New lines can be added at any time by typing the line number followed by the text.

(ii) *Deleting Lines*: If you wish to delete line 10, for example, type

```
DEL 10
```

To delete all lines between 10 and 100, inclusive, type

```
DEL 10-100
```

To delete all lines in the workspace, type

```
CLEAR
```

(iii) *Replacing Lines*: A line can be replaced simply by retyping it.

### 1.2.8 Sign-off Procedure

To end your session, type BYE. Your workspace is lost.

## 1.3 Examples

The remaining pages of this chapter contain annotated examples which illustrate several features and capabilities of the Waterloo microBASIC programming language.

**Example 1 Line Numbers, Comments, Variables, PRINT**

*Function*

This example sets a BASIC variable x equal to 12, squares this value placing the result in y, and prints both x and y.

```
10 ! Example 1
20 !
30 x = 12
40 y = x*x
50 print x,y
60 stop
```

Notes:

1. Each BASIC statement starts with a *line number*. These numbers are in ascending sequence and have values ranging from 1 to 65529.

2. Line numbers 10 and 20 contain *comments* or *remarks* which are indicated by the exclamation mark (!). The computer ignores remarks at execution time.

3. x and y are *BASIC variables*. These variables begin with a letter and can contain letters, digits and the underscore character. They can be up to 31 characters in length.

4. The PRINT statement contains a *print list*; in this case x and y. The *values* contained in x and y are printed side by side on the screen.

5. Multiplication is denoted by an asterisk (*) symbol. The arithmetic operators are:

| | |
|---|---|
| + | add |
| - | subtract |
| * | multiplication |
| / | divide |
| ↑ | exponentiate |

6. Expressions can be contained in parentheses:

   e.g. Y=(X+3.2)*6.4

7. The order of priority of operators is as in algebra. Evaluation of operations with equal priority proceeds from left to right.

**Example 2 Infinite Loops**

*Function*

The computations of Example 1 are placed in a *loop* which causes them to be repeated endlessly. When you run this program, it must be stopped by depressing the "STOP" key one or more times.

```
10 ! Example 2
20 !
25 loop
30 x = 12
40 y = x*x
50 print x,y
55 endloop
60 stop
```

Notes:

1. When the BASIC processor encounters the LOOP statement it indicates that all instructions that follow, up to the ENDLOOP statement, are contained in a *loop* and are to be repeated.

2. We learn how to stop loops automatically in later examples.

### Example 3 Indentation, Separators

*Function*

This example is identical to Example 2. However, all statements in the loop are *indented* three spaces to accent those which are to be repeated.

```
10 ! Example 3
20 !
25 loop
30     x = 12
40     y = x*x
50     print x,y
55 endloop
60 stop
```

Notes:

1 All statements can have as many leading blanks as are desired.

2. Blanks appear *within* statements as *separators*. For example, the blank between PRINT and x not only makes the program more readable to the human, but is necessary for proper understanding of the statement by the BASIC processor.

3. Other separators are the operators such as + - * / etc., and special separators include the ) ( = and ,. No blanks are necessary before or after these separators. Thus, x=12 need not be written as x = 12, but it is legal to do so.

**Example 4 Initial Value Outside Loops**

*Function*

Here x=12 is placed outside the loop and a new statement is inserted to increase x by 1 within the loop, thus causing a *table* of squares to be printed. Note that the program is in an endless loop which must be terminated with the "STOP" key.

```
10 ! Example 4
20 !
30 x = 12
35 loop
40     y = x*x
50     print x,y
52     x = x+1
55 endloop
60 stop
```

Notes:

1. The statement x=12 gives an *initial* value to x, and is not in the loop.

2. The statement x=x+1 causes the value of x to increase by 1 each time through the loop.

**Example 5 Exiting Infinite Loops, Relational Expressions**

*Function*

This example prints squares of the integers from 12 to 20, automatically terminating when x is equal to 21.

```
10 ! Example 5
20 !
30 x = 12
35 loop
40     y = x*x
50     print x,y
52     x = x+1
53     if x=21 then quit
55 endloop
60 stop
```

Notes:

1. The IF - QUIT statement

   ```
   if x=21 then quit
   ```

   causes the BASIC processor to end repetition of the loop if x has the value 21 when the IF is executed. Control of statement execution is transferred to the statement following the ENDLOOP.

2. The IF - QUIT statement can appear anywhere in the loop.

3. The x=21 in the IF - QUIT statement is called a *relational expression* and has the value *true* or *false*. The equal sign, =, is a *relational operator*. The relational operators are as follows:

   | | |
   |---|---|
   | = | equal to |
   | > | greater than |
   | >= | greater than or equal to |
   | < | less than |
   | <= | less than or equal to |
   | <> | not equal to |

**Example 6 Expanding Previous Examples**

*Function*

This example simply expands the previous examples to include the cube of the integers from 12 to 20, inclusive.

```
10 ! Example 6
20 !
30 x = 12
35 loop
40     y = x*x
45     z = x*x*x
50     print x,y,z
52     x = x+1
53     if x=21 then quit
55 endloop
60 stop
```

Notes:

1. A third variable z is introduced.

2. The statement z=x*x*x could have been written as z=x*y.

**Example 7 Character Strings, Print Zones**

*Function*

This example introduces a heading for the previous example.

```
10 ! Example 7
20 !
25 print 'x','y','z'
30 x = 12
35 loop
40     y = x*x
45     z = x*x*x
50     print x,y,z
52     x = x+1
53     if x=21 then quit
55 endloop
60 stop
```

Notes:

1. The 'x', 'y' and 'z' are called *character string constants* and will print *literally* as they appear in the program. Character string constants always contain a "string" of characters between quotes.

   e.g. 'ABCD:F'

2. The character string cannot contain a quote, but can contain any other legal character.

3. When you run this program, you will not like the spacing. The headings do not appear above the columns. This is because printing is done normally in 16 character *zones*. Character strings and numbers are printed *left-justified* in the zone, however, numbers are printed with a space for a sign in front. Negative numbers have a minus sign in this space. The next example corrects the problem.

**Example 8 Blanks in Character Strings**

*Function*

This example operates exactly as the previous one except that the headings are centered over the columns.

```
10 ! Example 8
20 !
25 print ' x',' y',' z'
30 x = 12
35 loop
40     y = x*x
45     z = x*x*x
50     print x,y,z
52     x = x+1
53     if x=21 then quit
55 endloop
60 stop
```

Notes:

1. The character string constants contain a number of *blank characters* which have the effect of shifting the x, y and z to the center of the column.

**Example 9 Square Root Function (SQR)**

*Function*

This example calculates a table of square roots of x for x having integer values from 1 to 30, inclusive.

```
 10 ! Example 9
 20 !
 30 print 'This is a table of Square Roots'
 40 x = 1
 50 loop
 60     y = sqr( x )
 70     print x,y
 80     x = x+1
 90     if x=31 then quit
100 endloop
110 stop
```

Notes:

1. The SQR is known as a *built-in* or *intrinsic function* and will compute the square root of the quantity in parentheses, provided the quantity is not negative.

2. Other intrinsic functions such as SIN and COS are available in BASIC.

3. Results are printed rounded, with at most 9 digits.

**Example 10 Sine/Cosine Functions (SIN,COS)**

*Function*

```
 10 ! Example 10
 20 !
 30 print 'This is a table of SINES and COSINES'
 40 print '   x',' SINx',' COSx'
 50 x = 0
 60 loop
 70     y = sin( x )
 80     z = cos( x )
 90     print x,y,z
100     x = x + .1
110     if x=3.1 then quit
120 endloop
130 stop
```

Notes:

1. When you run this program, it does not stop and must be terminated using the "STOP" key. This is because numbers are stored somewhat inaccurately by computers. For example, the fraction "one third" is written as .3333333 to seven figures in decimal notation; this is slightly incorrect. Computers work internally in *binary* notation and similar slight inaccuracies occur. Thus, 3.1 is probably stored something like

   3.09999999

   Since x=3.1 is never true, the program does not terminate. To correct this problem, replace line 110 with

   ```
   110   if x>3 then quit
   ```

   and run the program again.

**Example 11 PI Function, Scientific Notation**

*Function*

This example computes a table of sines and cosines of x for x ranging between PI/2 and PI radians in increments of 1.

```
 10 ! Example 11
 20 !
 30 print 'This is a table of SINES and COSINES'
 40 print '   x',' SINx',' COSx'
 50 x = pi/2
 60 loop
 70     y = sin( x )
 80     z = cos( x )
 90     print x,y,z
100     x = x + .1
110     if x>pi then quit
120 endloop
130 stop
```

Notes:

1. The output has an untidy appearance due to the use of *scientific notation*. In this notation, the numbers are expressed with an exponent.

   e.g. 1.23456789E12
   means 1.23456789 times 10 to the 12th power

2. PI is a known constant to BASIC, therefore, a value need not be assigned to it in the program. True mathematical PI, 3.141592653589793..., is a constant with an infinite number of decimal places. The microcomputer is capable of representing only a nine-digit approximation of this value. Consequently, COS( PI/2 ), as calculated by the computer, is a very small number (approximately .0000000007) instead of zero.

**Example 12 Input**

*Function*

This program is an endless loop which requests the user to "Type in x", and it returns the value of the cube of x.

```
 10 ! Example 12
 20 !
 30 print 'Test simple INPUT from keyboard'
 40 loop
 50     print 'Type in x'
 60     input x
 70     y = x*x*x
 80     print x, 'cubed =',y
 90 endloop
100 stop
```

Notes:

1. The purpose of the statement

   ```
   print 'Type in x'
   ```

   is to *prompt* the user; that is, this statement reminds the user that a value must be typed into the keyboard, followed by carriage return (RETURN key).

2. The statement

   ```
   print x, 'cubed=',y
   ```

   shows that character string constants can be placed along with variables in the print list.

3. This program is an endless loop and must be terminated with the "STOP" key.

**Example 13 Exiting a Program due to an Input**

*Function*

This program operates as Example 12 except that when you type -999 for x, the loop is terminated.

```
 10 ! Example 13
 20 !
 30 print 'Test simple INPUT from keyboard'
 40 loop
 50     print 'Type in x'
 60     input x
 65     if x = -999 then quit
 70     y = x*x*x
 80     print x, 'cubed =',y
 90 endloop
100 stop
```

**Example 14 Nested Loops, QUIT Limitations**

*Function*

This example computes a set of tables of squares of x, with the starting value of x, increment of x, and number of entries being prompted as input from the keyboard.

```
 10 ! Example 14
 20 !
 30 loop
 40     print 'Start x at'
 50     input x
 60     if x = -999 then quit
 70     print 'Vary x by'
 80     input xvary
 90     print 'Number of values for x is'
100     input n
110     print ' x',' x*x'
120     loop
130         y = x*x
140         print x,y
150         x = x + xvary
160         n = n - 1
170         if n=0 then quit
180     endloop
190     print 'Job finished'
200 endloop
210 stop
```

Notes:

1. This example introduces a "loop within a loop". The *inside loop* is the five statements between the inside LOOP - ENDLOOP pair. The *outside loop* contains all those statements between the outside LOOP - ENDLOOP pair, which *includes* the inner loop.

2. The QUIT statement exits from the loop immediately containing it only. The QUIT in line 170 transfers control to line 190.

3. To terminate, type -999 when the initial value of x is requested.

**Example 15 IF...ELSE...ENDIF**

*Function*

In this example, the user is asked to input a "dividend" and a "divisor". The "quotient" is printed, unless the divisor is zero, in which case an appropriate message is printed. To terminate, type -999 when the dividend is requested.

```
 10 ! Example 15
 20 !
 30 loop
 40     print "Input dividend"
 50     input dividend
 60     if dividend = -999 then quit
 70     print "Input divisor"
 80     input divisor
 90     if divisor = 0
100         print "Divisor = 0"
110     else
120         quotient = dividend / divisor
130         print quotient
140     endif
150 endloop
160 stop
```

Notes:

1. The purpose of this example is to illustrate the use of the IF - ELSE- ENDIF construction.

   a. The IF statement always contains a relational expression which, when evaluated, is true or false.

   b. If the relational expression is true, all statements between the IF and ELSE are executed.

   c. If the relational expression is false, all statements between the ELSE and ENDIF are executed.

2. The ELSE statement can be omitted. If so, all statements between the IF and ENDIF are executed when the relational expression is true.

3. IF's can be *nested*, if desired; that is, an IF-ELSE-ENDIF construction can contain other similar constructions.

**Example 16 Concatenation**

*Function*

The user is prompted to type in his first name, followed by his last name. The computer "composes" the two names into a single string and prints it.

```
10 ! Example 16
20 !
30 print "Illustrate CONCATENATION of strings"
40 loop
50     print "What is your first name"
60     input firstname$
70     if firstname$ = "quit" then quit
80     print "What is your last name"
90     input lastname$
100    fullname$ = firstname$ + " " + lastname$
110    print "Your full name is ", fullname$
120 endloop
130 stop
```

Notes:

1. The two names are assigned to two string variables firstname$ and lastname$. The dollar symbol, $, at the end of a name indicates that it is a string name.

2. The + operator causes the two strings before and after it to be combined into one string (concatenation), with no space between them. Thus, in order to have a blank space between the two names, it is necessary to "add" three strings together, with the center one being a character string constant containing a single blank.

**Example 17 Substring Operation**

*Function*

This example assigns the value ABCDEFGHIJ to the character variable x$, then *extracts* 4 characters from within the string, beginning at the third character. This new string is assigned to the string variable y$, and then printed.

Then, the program prints all the characters, one at a time.

```
 10 ! Example 17
 20 !
 30 x$ = "ABCDEFGHIJ"
 40 y$ = x$( 3 : 6 )
 50 print y$
 60 n = 1
 70 loop
 80     y$ = x$( n : n )
 90     print y$
100     n = n + 1
110     if n=11 then quit
120 endloop
130 stop
```

Notes:

1. The substring operation is defined as follows:

   character variable( M : N )

   The operation causes the Mth through the Nth characters from the character variable to form a new string. If M=N then the string consists of one character (the Mth). If M>N or N<=0 or M > string-length then the result is the "null" or empty string.

2. One can also assign a character string to a substring. To illustrate this try the following assignment and print the new value of x$.

   x$( 5 : 5 ) = 'eee'

**Example 18 Expand on Example 17**

*Function*

The character string ABCDEFGHIJ is assigned to x$. The first print line will contain the first character, the second print line will contain the first two characters, etc., with the 10th line containing all 10 characters in the string.

```
 10 ! Example 18
 20 !
 30 x$ = "ABCDEFGHIJ"
 40 n = 1
 50 loop
 60     y$ = x$( 1 : n )
 70     print y$
 80     n = n + 1
 90     if n=11 then quit
100 endloop
110 stop
```

**Example 19 Length Function (LEN)**

*Function*

The user is asked to type in his name. The program prints the name vertically, one character per line.

```
 10 ! Example 19
 20 !
 30 loop
 40     print "What is your name"
 50     input x$
 60     if x$ = "quit" then quit
 70     n = len( x$ )
 80     t = 1
 90     loop
100         y$ = x$( t : t )
110         print y$
120         if t=n then quit
130         t = t + 1
140     endloop
150 endloop
160 stop
```

Notes:

1. As each name will be of different length, we use the LEN function.

   LEN(character variable)

   returns the number of characters in the string.

2. The program stops when the characters "quit" are entered as a name.

**Example 20 Expands Previous Examples**

*Function*

This example asks the user to input a three letter word. The program prints the word with the letters in reverse order.

```
 10 ! Example 20
 20 !
 30 loop
 40     print "What is the three-letter name"
 50     input x$
 60     if x$ = "quit" then quit
 70     y1$ = x$( 3 : 3 )
 80     y2$ = x$( 2 : 2 )
 90     y3$ = x$( 1 : 1 )
100     backname$ = y1$ + y2$ + y3$
110     print backname$
120 endloop
130 stop
```

**Example 21 VALUE and VALUE$**

*Function*

The character string '1234567', is converted to numeric, 2 is added and the result is printed. A number 123 is converted to a string and is concatenated to 'CAT' to produce the string '123CAT', which is printed.

```
 10 ! Example 21
 20 !
 30 x$ = "1234567"
 40 y = value( x$ )
 50 y = y + 2
 60 print y
 70 x = 123
 80 y$ = value$( x ) + "CAT"
 90 print y$
100 stop
```

Notes:

1. The intrinsic function

   VALUE(character variable)

   converts the value of the character variable to a numeric quantity. Note that the string must contain only characters that form a valid representation of a numeric constant value (e.g., 1.2 or -5).

2. The intrinsic function

   VALUE$(numeric variable)

   converts the value of the numeric variable to a string which contains characters representing the numeric value as it would be displayed with the PRINT statement.

**Example 22 Matrices, Printing a Blank Line**

*Function*

This example requests the user to input exactly 10 names. It then prints them back in reverse sequence.

```
 10 ! Example 22
 20
 30 dim x$( 10 )
 40 i = 1
 50
 60 loop
 70       print 'Name please'
 80       input a$
 90       x$( i ) = a$
100       i = i + 1
110       if i=11 then quit
120 endloop
130
140 i = 10
150 print
160 print 'The names in reverse order'
170
180 loop
190     print x$( i )
200     i = i - 1
210     if i = 0 then quit
220 endloop
230
240 stop
```

Notes:

1. The names are stored in a *matrix* which is set up using the DIM statement in line 30. This statement creates an *eleven* element array, with the elements being

   x$(0),x$(1),x$(2), ... ,x$(10)

   Notice that element x$(0) is not used in this program.

2. Each element of the matrix is able to hold a character string, with each string being of a different length, if desired.

3. Numeric matrices can be defined with a similar DIM statement. Note that numeric names do not end with a dollar symbol, $.

4. Matrices can have as many *dimensions* as are required.

5. Line 150 causes a blank line to be printed.

**Example 23 ';' in a PRINT Statement**

*Function*

The user inputs 10 names. Then he is asked for an integer between 1 and 10. If he enters 5, the 5th name is printed, etc.

```
 10 ! Example 23
 20
 30 dim x$( 10 )
 40 i = 1
 50
 60 loop
 70     print 'Name please'
 80     input a$
 90     x$( i ) = a$
100     i = i + 1
110     if i = 11 then quit
120 endloop
130
140 loop
150     print 'Enter a number between 1 and 10'
160     input i
170     if i = -1 then quit
180     print 'Name '; i; ' is '; x$( i )
190 endloop
200
210 stop
```

Notes:

1. The semicolons, ;, used in line 180 indicate that the "16 character print zone" is to be ignored and each value is printed immediately following the previous value.

**Example 24 Matrix Element Zero**

*Function*

This program is identical to the previous example except that 11 names are input, utilizing the 0th element of the matrix x$.

```
 10 ! Example 24
 20
 30 dim x$( 10 )
 40 i = 0
 50
 60 loop
 70     print 'Name please'
 80     input a$
 90     x$( i ) = a$
100     i = i + 1
110     if i = 11 then quit
120 endloop
130
140 loop
150     print 'Enter a number between 0 and 10'
160     input i
170     if i = -1 then quit
180     print 'Name '; i; ' is '; x$( i )
190 endloop
200
210 stop
```

**Example 25 OPTION BASE**

*Function*

The program once again asks the user for 10 names. However, only 10 elements are allocated since we use OPTION BASE 1.

```
 10 ! Example 25
 20
 25 option base 1
 26
 30 dim x$( 10 )
 40 i = 1
 50
 60 loop
 70     print 'Name please'
 80     input a$
 90     x$( i ) = a$
100     i = i + 1
110     if i = 11 then quit
120 endloop
130
140 loop
150     print 'Enter a number between 1 and 10'
160     input i
170     if i = -1 then quit
180     print 'Name '; i; ' is '; x$( i )
190 endloop
200
210 stop
```

Notes:

1. In line 25 we introduce the statement

   OPTION BASE 1

2. This causes *all* matrices in the program to begin with the first element having an index of 1 rather than zero. Thus, the matrix x$ contains 10 elements rather than 11.

**Example 26 Procedures**

*Function*

This example invites the user to submit 10 names which are stored in an matrix and then are printed in reverse order.

```
 10 ! Example 26
 20
 30 dim x$( 10 )
 40 call read_names
 50 print "Names in reverse order"
 60 call print_names
 70 stop
 80
 90 proc read_names
100 i = 1
110 loop
120     print "Name please"
130     input a$
140     x$( i ) = a$
150     i = i + 1
160     if i = 11 then quit
170 endloop
180 endproc
190
200 proc print_names
210 i = 10
220 loop
230     print x$( i )
240     i = i - 1
250     if i = 0 then quit
260 endloop
270 endproc
```

Notes:

1. The main purpose of this example is to introduce the procedure definition facility in BASIC. We define two procedures; namely, read_names and print_names.

2. Procedure names are defined with the PROC statement as illustrated in lines 90 and 200.

3. The procedure definition is the group of statements between the PROC and ENDPROC statements.

4. The procedure is invoked by including it in a CALL statement. For example

```
call read_names
```

causes the read_names procedure to be executed. This use of a procedure call permits one to *modularize* the program and is a matter of programming *style*.

**Example 27 Procedures with Parameters**

*Function*

This example is similar to Example 26 but permits the user to indicate the number of names which are to be entered.

```
 10 ! Example 27
 20
 30 dim x$( 10 )
 34 print "How many names"
 36 input n
 40 call read_names( n )
 50 print "Names in reverse order"
 60 call print_names( n )
 70 stop
 80
 90 proc read_names( count )
100 i = 1
110 loop
120     print "Name please"
130     input a$
140     x$( i ) = a$
150     i = i + 1
160     if i > count then quit
170 endloop
180 endproc
190
200 proc print_names( count )
210 i = count
220 loop
230     print x$( i )
240     i = i - 1
250     if i = 0 then quit
260 endloop
270 endproc
```

Notes:

1. The purpose of this example is to introduce a *parameter* in the procedure. The PROC statement now contains not only the procedure name, but a parameter list "(count)". The parameter variable, count, is assigned the value of variable n when the procedure is called in line 40. The value of n is established in line 36 when the user types a number.

2. When the procedures are called in lines 40 and 60, the parameter list must be included with the procedure name.

3. There can be as many parameters in the list as are required. Commas are used to separate multiple parameters in a list.

**Example 28 User Functions**

*Function*

This example permits the user to input up to 10 numbers, then computes and prints the average of these numbers.

```
 10 ! Example 28
 20
 30 dim x( 10 )
 40 print "How many numbers"
 50 input n
 60 call read_numbers( n )
 70 print "Average of numbers"
 80 print fn_average( n )
 90 stop
100
110 proc read_numbers( count )
120 i = 1
130 loop
140     print "Number please"
150     input number
160     x( i ) = number
170     i = i + 1
180     if i > count then quit
190 endloop
200 endproc
210
220 def fn_average( count )
230 i = 1
240 total = 0
250 loop
260     total = total + x( i )
270     i = i + 1
280     if i > count then quit
290 endloop
300 fn_average = total / count
310 fnend
```

Notes:

1. The purpose of this example is to introduce the function definition facility in BASIC. We define a function named "fn_average". The function has one parameter, count, which performs a similar role to the parameter used with procedures in the previous example.

2. Function names are defined with a DEF statement as illustrated in line 220. The function definition is the group of statements between the DEF and FNEND statements.

3. The primary difference between functions and procedures is that function names are assigned a value within the function definition, as illustrated in line 300. If this was not done, the function name fn_average would be assigned the value zero when the function completed its work.

4. The function is *invoked* by including the function name in a statement where an expression or value can appear, as illustrated in line 80.

5. Note that all function names must begin with fn or FN. Functions can be numeric, string or integer functions. String function names always end with a $. Integer function names end with a % (integers are discussed in later examples).

**Example 29 Files, OPEN, CLOSE, IO_STATUS, ON EOF**

*Function*

This program creates a file on disk called "namefile". The program requests the user to type names at the terminal. These names are printed as records on the disk file. When the name quit is entered, the program halts.

```
 10 ! Example 29
 20
 30 on eof ignore
 40 open #2,'namefile',output
 50
 60 loop
 70     print 'Name'
 80     input name$
 90     if name$ = 'quit' then quit
100     print #2, name$
110 endloop
120
130 close #2
140 open #2,'namefile',input
150
160 loop
170     input #2, name$
180     if io_status <> 0 then quit
190     print name$
200 endloop
210
220 close #2
230 stop
```

Notes:

1. In statement 40 we *open* the file. If there is no such file as namefile, a new file is automatically created. We indicate that the file will be used as output; thus if this is not a new file, all old information will be destroyed. The #2 in the open statement is a unit number assigned to the file *for this program only*. All other references to the file in this program are done using this number rather than the file name.

2. The file name (e.g., namefile) is always a character string of 16 or fewer characters.

3. When we want to write data into the file, we use a statement like line 100. This causes a "record" containing the single field name$ to be written onto unit #2.

4. When the file has been written, it must be *closed* as in line 130. A special "end of file indicator" is written at the end of the file.

5. When the file is reopened for *input* in line 140, it is automatically positioned at the *beginning*.

6. As the records are read, one name at a time is assigned to the variable name$ in line 170.

7. When the "end of file indicator" has been read, the built-in function IO_STATUS will have a non-zero value. Thus, we use this as a test for termination.

8. The built-in function IO_STATUS reflects the status of the latest input/output operation performed. The "system" will automatically alter the value of IO_STATUS when an INPUT or PRINT encounters special conditions such as end-of-file or an error.

9. In line 30 we have the statement

   ON EOF IGNORE

   This is needed to prevent the system from taking automatic action when end of file occurs.

**Example 30 Multiple Field File Records**

*Function*

This program creates two fields in each record in the file "namefile".

```
 10 ! Example 30
 20
 30 on eof ignore
 40 open #2,'namefile',output
 50
 60 loop
 70     print 'Input name and age'
 80     input name$, age
 90     if name$ = 'quit' then quit
100     print #2, name$ + ',', age
110 endloop
120
130 close #2
140 open #2,'namefile',input
150
160 print 'Age','Name'
170 loop
180     input #2, name$, age
190     if io_status <> 0 then quit
200     print age, name$
210 endloop
220
230 close #2
240 stop
```

Notes:

1. Each time line 80 is executed you will need to type a name and age. These must be separated by a comma or entered on separate lines.

2. Each time line 100 is executed, both NAME$ and AGE are written to the file, collected into 16 character "windows" as they would appear on the screen. A comma is concatenated to NAME$ to separate the values for subsequent input.

3. When the data is read in line 180, we use *exactly* the same number and type of fields as when the data was written.

4. Note that in line 200 we print age to the left of name to show that this ordering is independent of the order of the fields in the record.

5. To terminate, you must type quit and some numeric value, as the INPUT statement is expecting two values to be entered.

**Example 31 Files using LINPUT**

*Function*

Here we create a file, "namefile", and subsequently read each record as a character string with the LINPUT statement.

```
 10 ! Example 31
 20
 30 on eof ignore
 40 open #2,'namefile',output
 50
 60 loop
 70     print 'Input name and age'
 80     input name$, age
 90     if name$ = 'quit' then quit
100     print #2, name$,'--';age
110 endloop
120
130 close #2
140 open #2,'namefile',input
150
160 loop
170     linput #2, a$
180     if io_status <> 0 then quit
190     print a$
200 endloop
210
220 close #2
230 stop
```

Notes:

1. Line 170 introduces the input statement LINPUT. This causes the *entire record* to be read and assigned to the character variable a$. When a$ is printed, we see the image of each record as it was printed to the file in line 100.

2. LINPUT allows us to read an entire *record* into a character variable. INPUT allows us to read the record by specifying *all* of its fields.

**Example 32 Random Number Generator**

*Function*

This program prints 20 random numbers between 0 and 1.

```
 10 ! Example 32
 20
 30 randomize
 40 i = 1
 50 loop
 60     print rnd
 70     i = i + 1
 80     if i = 21 then quit
 90 endloop
100 stop
```

Notes:

1. The function RND causes a random number to be generated and returned.

2. The statement in line 30; namely, RANDOMIZE, causes the random number generator to be "seeded" with a special starting value.

**Example 33 Integer-Part Function (IP)**

*Function*

This program produces 20 random integers in the range 0 to 9, inclusive.

```
 10 ! Example 33
 20
 30 randomize
 40 i = 1
 50 loop
 60     print ip( rnd * 10 )
 70     i = i + 1
 80     if i = 21 then quit
 90 endloop
100 stop
```

Notes:

1. The built-in function IP is used to obtain the *integer part* of the argument. Thus, IP(1.8) returns 1 and IP(96.74) returns 96.

2. We multiply RND by 10 to bring the value into the range from 0 to 10, not including 10.

**Example 34 Integer Variables**

*Function*

This program performs the same function as the previous example making use of an integer variable in place of using the IP function.

```
 10 ! Example 34
 20
 30 randomize
 40 i = 1
 50 loop
 60     n% = rnd * 10
 65     print n%
 70     i = i + 1
 80     if i = 21 then quit
 90 endloop
100 stop
```

Notes:

1. Integer variables are used to represent "whole numbers" whereas normal numeric variables can represent whole or fractional numbers.

2. Integer variable names end with a % character (for example, n% used in lines 60 and 65).

3. Integer variables can be used for values in the range from -32768 to +32767.

**Example 35 Integer Arithmetic**

*Function*

This program illustrates a peculiarity of integer arithmetic compared to normal arithmetic.

```
 10 ! Example 35
 20
 30 i% = 1
 40 two% = 2
 50 loop
 60     print i%, i% / 2, i% / two%
 70     i% = i% + 1
 80     if i% = 11 then quit
 90 endloop
100 stop
```

Notes:

1. An arithmetic operation, such as i%/two%, which has *only* integer variables as operands, is evaluated according to the values of integer arithmetic and produces an integer result.

2. An arithmetic operation, such as i%/2, which has both an integer variable and a normal numeric value, is evaluated according to normal arithmetic rules and produces a normal numeric value.

3. The division operations performed in line 60 illustrate one of the major differences between normal and integer arithmetic. In the case of integer division, fractional results are truncated to the integer part of the value.

**Example 36 Integer Arithmetic**

*Function*

This program distinguishes between even and odd integer values making use of the peculiarities of integer division described in the previous example.

```
 10 ! Example 36
 20
 30 i% = 1
 40 two% = 2
 50 loop
 60     if ( i% / two% ) * two% = i%
 70         print i%, 'even'
 80     else
 90         print i%, 'odd'
100     endif
110     i% = i% + 1
120     if i% = 11 then quit
130 endloop
140 stop
```

Notes:

1. The result of division by integer 2 is multiplied by integer 2 and compared with the original integer. Odd numbers produce fractional results truncated to integer which do not match the original values when multiplied and compared.

2. Similar techniques can be used to compute the modulus of integer values using different number bases.

# Waterloo microBASIC

# Command Language Reference Guide

# Waterloo Computing Systems Newsletter

The software described in this manual was implemented by Waterloo Computing Systems Limited. From time-to-time enhancements to this system or completely new systems will become available.

A newsletter is published periodically to inform users of recent developments in Waterloo software. This publication is the most direct means of communicating up-to-date information to the various user. Details regarding subscriptions to this newsletter may be obtained by writing:

**Waterloo Computing Systems Newsletter**
**Box 943,**
**Waterloo, Ontario, Canada**
**N2J 4C3**

# Chapter 1

# Commands

## 1.1 Introduction

*Commands* may be issued to the Waterloo microBASIC system by typing them at the keyboard. Generally speaking, the commands recognized by the system are used to prepare programs, modify programs and store or retrieve copies of programs.

*Abbreviated forms* of most commands can be used. For the sake of clarity, abbreviations are not used in introducing these commands. An appendix of this manual contains a command summary which shows acceptable command abbreviations.

## 1.2 Starting and Finishing

When your personal computer's power is turned on, a message is displayed requesting that you select a processor. To select BASIC, type the letter b and press the RETURN key. A heading will be displayed indicating that Waterloo microBASIC has been activated.

In this initial state the BASIC workspace in the computer is empty. At this point a program can be entered as will be described in subsequent chapters. Whenever you have completed working with one program and wish to enter a new one, you should type the command

CLEAR

to re-initialize the workspace to the empty state. Commands are shown here in uppercase, but can be entered in either upper or lower case (e.g., CLEAR or clear).

When you are finished working with BASIC and wish to use another processor, type the command

BYE

to return to processor selection level. A message will be displayed requesting selection of a processor, similar to when you turned on the computer's power.

Of course, if you are finished working with the computer and do not wish to use another processor, you can simply turn off the power and need not type the BYE command.

# Chapter 2

# Entering a Program

A program consists of lines which contain statements. Each line has a number on the front. This number must be a whole number (integer) and can be in the range of 1 through 65529.

When a line with a number on the front is typed, it is recognized to be a line of a program and is not treated as a command. Lines of a program are kept in the computer workspace in order of *line number*. The statements of a program are also executed in order of line number in normal circumstances.

Regardless of the order in which lines are typed, BASIC always maintains them in ascending order of line number. The following example shows a program with line numbers 1 through 6. Whether these lines were typed in order 1,2,3,4,5,6 or 2,5,4,3,6,1 the result would be identical.

```
1  !  Example Program
2  !  Calculate Fahrenheit temperature from Centigrade
3  C = 100
4  F = (C * 9)/5 + 32
5  print C,F
6  stop
```

Normally, lines are typed more or less in order by line number. Frequently, however, it is necessary to go back and type a line between two existing lines to correct an error of omission or simply to enhance the function of a program.

It would be difficult to add lines to the previous example, except at the end of the program, since no line numbers were left unused in the range 1 through 6. It is common practice, therefore, to leave gaps between line numbers as illustrated in the following example.

```
10  !  Example Program
20  !  Calculate Fahrenheit temperature from Centigrade
30  C = 100
40  F = (C * 9)/5 + 32
50  print C,F
60  stop
```

If a line is entered with a line number that matches an existing line in a program, the new line replaces the existing one. For example, if the line

```
50  print F,C
```

was entered when the above program was in the workspace, the old line 50 would be replaced and the program henceforth would display the values of F and C in the new order. This is an example of *editing* a program. Further means of editing programs are introduced in a later chapter.

This means of line replacement can be convenient, but is also somewhat dangerous. It is easy to inadvertently replace lines when entering programs this way. For example, instead of adding a line 300 to a program, it is easy to miss a digit and replace line 30.

Another facility is provided for entering lines of program which reduces the probability of such errors and eliminates the tedium of manually typing line numbers. The AUTOLINE command automatically generates line numbers for program entry. The generated number for a line is displayed and the rest of the line can be typed. When one line is completed and entered, the next line number is displayed. Automatic line number generation is terminated by pressing the STOP key or by erasing the generated number and pressing RETURN.

The line number to start automatic generation and the amount to increment for each new line may be specified as follows:

AUTOLINE *start#, increment*

where start# is a valid line number and increment is a suitable integer value. If no start# is specified, the first generated line numbers will be 10 greater than the highest line number of the program in the workspace. If the workspace is empty, the default start# is 10. If no increment is specified, a default of 10 is assumed.

For example,

| | |
|---|---|
| AUTOLINE | generates lines 10, 20, 30, .... |
| AUTOLINE 110 | generates lines 110, 120, 130, .... |
| AUTOLINE 230, 5 | generates lines 230, 235, 240, .... |

Lines added to a program with AUTOLINE are handled in the same manner as lines with manually typed line numbers. Thus, lines with numbers between existing line numbers will be inserted between the existing lines, and lines with numbers matching existing line numbers will replace existing lines. Although inadvertent replacement of existing lines is less likely with AUTOLINE, caution must be exercised to avoid this.

# Chapter 3

# Running a Program

When a program has been entered into the computer workspace, the command

RUN

can be entered to execute or run it. The computer begins processing at the lowest numbered line and performs the actions specified by executable statements. Statements are normally executed *sequentially*, in order of ascending line number, however, this order can be altered by *control statements* in the program. Control of statement execution in a program is described in the Programming Language portion of this manual.

The program will normally execute until it is *terminated* by a STOP or END statement. If the end of the program is encountered, the program is terminated as if an END statement appeared there. The program will be *interrupted* if an error is encountered, or it can be intentionally interrupted by pressing the STOP key.

Before executing a program, the BASIC system examines the program's statements, resolving information concerning control structures. If control structures are found to be incomplete or invalid, diagnostic messages are displayed pointing out the problems and the program is not run.

If no problems are detected the message

Executing...

is displayed and program execution commences. When program execution ends, the message

Ready

is displayed indicating that the system is again ready to accept commands.

The RUN command initializes the program such that it executes each time exactly as if it had been freshly entered. This is true regardless of what state the program may be in as a result of previous program execution. That is, any numeric data items are set to zero, character data items are set to the empty or null string, data files are closed, and outstanding statement-control information is purged.

Other means of executing portions of programs and continuing interrupted execution without initialization are introduced in the chapter entitled "Debugging Programs".

# Chapter 4

# Editing a Program

Frequently it is necessary to change or edit a program that has been entered into the computer workspace. It has been explained that lines may be added to a program, or existing lines replaced, simply by typing the lines with appropriate line numbers or entering them with the AUTOLINE facility.

## 4.1 LIST Command

When editing, it is useful to be able to examine the current lines of a program. The command

```
LIST
```

provides a means of listing the lines of a program on the computer's display screen. This command, as shown, will display all of the lines of the program. If there are more lines to be listed than can be displayed on the screen, the LIST command pauses each time a screen is filled. LISTing continues if just the RETURN key is pressed. If another command is entered while LISTing is paused, LISTing terminates and the new command is processed.

In some cases you may wish to examine a single line. This can be accomplished with the form of the LIST command shown below

LIST line#

where line# is a valid line number. The following examples demonstrate commands to display lines 100 and 15 respectively.

```
LIST  100
LIST   15
```

The general form of the LIST command permits a range of lines to be displayed as shown below

LIST start#-end#

where start# is the first line number in the range and end# is the last line number in the range. If start# is not specified, it is assumed to be the first line of the program. Similarly, if end# is not specified, it is assumed to be the last line of the program. The following examples illustrate use of line ranges with the LIST command.

| | |
|---|---|
| LIST 100-200 | display lines 100 through 200 |
| LIST -200 | display from start of program through line 200 |
| LIST 100- | display from line 100 through end of program |
| LIST - | display all lines of the program |

Note that the last example is equivalent to the simple form of the LIST command with no line or line range specified.

## 4.2 DELETE Command

The DELETE command provides a means of removing lines from a program. A line number or line range must be specified with the DELETE command in the same manner as the LIST command. The following examples illustrate use of this command:

| | |
|---|---|
| DELETE 10 | delete line 10 |
| DELETE 10-40 | delete lines 10 through 40 |
| DELETE -40 | delete from start of program through line 40 |
| DELETE 900- | delete from line 900 through end of program |

## 4.3 Changing Lines

Any line of a BASIC program can be changed by retyping the entire line. Some computer systems, however, support screen editing features which allow changing of lines without retyping. The remainder of this section describes use of these features to change lines of a program on such systems.

Any line displayed on the screen can be modified and re-entered using special editing keys to insert, delete and replace characters. To accomplish this, position the cursor to the line you wish to modify with the "cursor-up" (upward arrow) key or "cursor-down" (downward arrow) key. Position the cursor within the line using the "cursor-right" (rightward arrow) key and "cursor-left" (leftward arrow) key. Characters can be replaced simply by typing the new characters. Pressing the "delete-character" key causes a character to be removed from the line. The "insert-character" key can be used to add characters in the middle of a line.

Once the line has been suitably modified on the screen, it may be entered to the computer by pressing the "RETURN" key. Note that only the line at which the cursor is positioned is entered into the computer. If changes to other lines of the screen display have been made without pressing the "RETURN" key for those lines, the modified lines will not have been entered to the computer.

This type of editing is typically used to change the contents of a line. However, it can also be used to make a copy of a line in another part of the program. This may be accomplished simply by altering the line number at the front to designate the position of the new copy, and pressing "RETURN".

Similarly, previously issued commands that remain on the screen can be re-issued, perhaps after slight modification, with this technique. Editing of the screen display and re-entry of lines in this manner provides a simple but powerful editing capability.

## 4.4 RENUMBER Command

As noted earlier, it is often necessary to add new lines to a program between existing lines. It was recommended that gaps be left between line numbers used to facilitate such additions later. Even where gaps are left, however, it is sometimes necessary to insert more lines than have been allowed for in the gaps. Faced with this type of situation, one quickly learns how to use the RENUMBER command.

RENUMBER changes the line numbers at the front of each program line without changing the relative order. In addition to changing the numbers on the front of each

line, it appropriately updates any primitive control statements in the program that refer to line numbers. Any references to non-existent lines are displayed for inspection, but are not changed.

The general form of this command is shown below:

RENUMBER start#, increment

where start# is the number to be generated for the first program line, and increment is the amount to add in generating each successive line number. An increment of ten is assumed if increment is not specified. If neither start# nor increment are specified, a starting line number of ten (10) and increment of ten is used.

The following examples illustrate the assignment of new numbers to the lines of a program.

| | |
|---|---|
| RENUMBER | assign numbers 10, 20, 30, .... |
| RENUMBER 100 | assign numbers 100, 110, 120, .... |
| RENUMBER 100, 20 | assign numbers 100, 120, 140, .... |

Thus, RENUMBER may be used to enlarge gaps for inserting lines between existing lines. Programmers often RENUMBER the lines of their program for aesthetic purposes, or for consistency before storing a copy for later use.

# Chapter 5

# Storing and Retrieving a Program

Once a program has been entered, tested and edited to the programmer's satisfaction, a copy of the program can be stored on an external storage medium such as a disk. This copy may be subsequently loaded back into the computer workspace to be used again.

As a program is entered, each line is compressed into an *encoded* form to make optimal use of the limited workspace available in the computer. When the program is displayed with the LIST command, each line is *decoded* as it is displayed so that it is presented in *source* form, that is, the form in which it was originally typed. Different commands are provided to store and retrieve copies of programs in both encoded and source forms for reasons that will be explained as the commands are introduced.

## 5.1 Files

Programs are stored on the disk as files. Each program file is identified by a name. In the case of the Commodore disk units used with this system, each file name can consist of up to 16 characters. For example, a program file could be named MYFIRSTPROGRAM.

Each disk *unit* has two *drives*. A flexible disk can be placed in each drive. The rightmost drive of a unit is designated as drive 0 and the leftmost as drive 1. The system will handle multiple disk units and consequently, each unit is identified by a number or *address*. When only one disk unit is attached to a system, its address is normally set to 8.

The term filename, as used in the description of commands in this manual, can consist of a simple name such as MYFIRSTPROGRAM. In this case, the file is assumed to reside on a disk with unit address 8 and drive 0. If the file in question does not conform to these assumptions, it is necessary to qualify the filename further. A description of general filenames is presented in the appendix entitled 'Files'. For our current purposes, use of simple names will suffice.

## 5.2 STORE Command

The command

STORE 'filename'

copies the program which currently resides in the computer workspace to the file specified. The copy is in encoded form just as the program is represented in the computer workspace. This form of program copy is advantageous for two reasons: the program in compressed format requires minimal space on the external medium; the entire program is copied in one step from the workspace, resulting in fast operation.

If a program is copied to the disk with the command

STORE 'MYFIRSTPROGRAM'

a new file is created with this name. If a program has previously been copied with the STORE command to a file with this name, the old program copy will be replaced when the STORE command copies the current program. If another type of file already exists with the filename used, the STORE command will fail and an appropriate diagnostic message will be displayed.

## 5.3 LOAD Command

The command

LOAD 'filename'

retrieves a program copy from a file into the computer workspace. The LOAD command retrieves copies created with the STORE command. Since the copy of the program is in encoded form, it can be transferred quickly into the workspace in a single step.

When a program is LOADed into the workspace, for example,

LOAD 'MYFIRSTPROGRAM'

the workspace is first initialized, clearing any program currently resident, before the program copy is transferred from the file.

## 5.4 SAVE Command

The command

SAVE line range 'filename'

copies a program, or portion of a program, from the computer workspace to the file specified. The copy is in *source* or *character* format; that is, program lines are decoded into the form in which they were originally typed. The file created appears exactly like a character data file, where the data is the source lines of the program.

A portion of a program may be SAVEd by specifying a line range just as with the LIST command introduced earlier. If no line range is specified, all lines of the program are SAVEd.

Since each program line is decoded and transferred individually, the SAVE operation can be considerably slower than STORE. In addition, SAVEd files generally occupy more space than STOREd files.

The SAVE command, however, provides some capabilities that cannot be accomplished with the STORE command. Since the lines of the program are SAVEd in character format, a copy of the program can be printed by specifying a filename of printer. In addition, portions of a program can be SAVEd in a disk file and later incorporated into other programs with the MERGE command. This technique

provides a convenient means of duplicating a common sequence of statements in several different programs.

The following examples illustrate use of the SAVE command.

| | |
|---|---|
| SAVE 'PROGRAMSOURCE' | copy all lines of the program to disk file PROGRAMSOURCE |
| SAVE 'printer' | print a copy of the program |
| SAVE 300-800 'COMMONLINES' | copy lines 300 through 800 of the program to disk file COMMONLINES |

If lines of a program are copied to a disk file with the SAVE command, and no files exist with the specified filename, a new character data file is created. If a character data file already exists with this filename, the old file is replaced. If another type of file already exists with the filename used, the SAVE command will fail and an appropriate diagnostic message will be displayed.

## 5.5 MERGE Command

The command

MERGE 'filename'

adds lines from the specified file to the current program in the workspace as if they were typed at the keyboard. The lines from the file are inserted into the program in the position indicated by the line number at the front of each line. If incoming lines have line numbers matching existing lines of the program, the existing lines are replaced.

The file specified must be a character data file and typically is created with the SAVE command. Each line of the file must have a valid line number at the front. If a line is encountered without a valid line number, the MERGE command quits and lines which follow are not copied into the workspace.

This facility is particularly useful when several programs perform some operations in common. The statements which perform these operations can be SAVEd in a file and MERGEd into other programs without being retyped. Care must be taken that the lines to be MERGEd fall in a line number range that does not conflict with the rest of the line numbers of the program. The RENUMBER command can be useful in sorting out such conflicts if they arise.

## 5.6 OLD Command

The command

```
OLD 'filename'
```

initializes the workspace, clearing any program currently resident, and then adds lines from the specified file to the workspace as if they were typed at the keyboard. Except that it first initializes the workspace, the OLD command operates exactly as the MERGE command.

The file specified must be a character data file (usually created by the SAVE command). The OLD command is useful for retrieving programs from files which may have been copied from other computer systems in character or *source* format.

Each line of the file is transferred, encoded and inserted into the workspace individually. Consequently, this operation is slower than LOADing a program from a file created with the STORE command. In general, it is recommended that STORE and LOAD commands be used to create and retrieve copies of programs when using a disk directly attached to your computer system. SAVE and OLD commands can be used to accomplish this function but are slower and the source representation of the program occupies more file space.

## 5.7 RUN Command

The command

```
RUN
```

was introduced in an earlier chapter. An extended form of this command

```
RUN 'filename'
```

will retrieve a copy of a program from the specified file and then execute it in the same manner as the simple RUN command. The file containing the program copy can have been created with the STORE command or can be a character data file as created by the SAVE command.

# Chapter 6

# Debugging Programs

Developing BASIC programs would be fairly straight-forward if they always performed as expected. Unfortunately, human beings frequently make mistakes when writing programs and when typing them at the keyboard. The process of finding errors in a program is know as *debugging*, that is, eliminating the "bugs". This process includes many different techniques for testing programs and locating errors. The tools provided by Waterloo microBASIC for debugging and some fundamentals of debugging methodology are introduced in this chapter. A strategy and style of debugging will be developed by the user as he or she gains experience.

## 6.1 Immediate Mode

Most statements in the BASIC programming language can be executed directly or immediately by typing them at the keyboard *without a line number*. For example, if you type

```
print 'Hello'
```

the word Hello will be displayed on the screen as soon as the statement is entered. This *immediate mode* of operation extends the command language available at the keyboard to include most statements of the programming language. In addition,

immediate mode statements can operate on the data items of the current program in the workspace. The role that immediate mode statements fulfill in debugging programs will be illustrated in the remaining sections of this chapter.

Generally speaking, any executable statement that is meaningful by itself is valid for immediate mode execution. Examples of statements that are not meaningful by themselves are LOOP and ENDLOOP. These two statements delimit a group of statements in one type of control structure available for repetitive execution of statements.

## 6.2 Interrupting Programs

When the microBASIC system recognizes an error as a program is executing, it will interrupt the program, display a diagnostic message indicating the type of error and then display the line that was executing when the error occurred. A mark is displayed underneath the program line indicating the position where the error was detected. When a program has been interrupted, commands can be entered, the program edited, and immediate mode statements executed. For example, a program might be interrupted because it attempted to input data from a file which had not been opened. In this case, an OPEN statement could be executed in immediate mode and execution could be allowed to continue to test the remainder of the program.

The type of errors recognized by microBASIC are, generally speaking, violations of rules or requests to perform operations which cannot be accomplished for one reason or another. Unfortunately, these are not the only types of problems that can arise with programs. It is very easy and very common to erroneously define a program that will carry on executing indefinitely, repeating the same valid sequence of operations. If the sequence being repeated contains no PRINT statements, nothing will appear on the screen regardless of how long you wait. Execution of such infinite loops can be interrupted manually by pressing the STOP key. When this is done, the microBASIC system displays a message and displays the line which was just about to be executed.

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 key data items to see if this gives you a clue. In particular, you should examine the values of any data items used in decisions which control repetition. For example, if a group of statements is to be repeated until the value of I becomes 100, type

```
print I
```

to immediately display the current value of I. If the value does not make sense,

examine the program to determine how the value of I is calculated.

In some cases it helps to alter the value of a data item using an immediate mode assignment statement, and then let the program continue executing to observe its behaviour.

## 6.3 Resuming Execution after Interruption

Some programs are expected to operate for several hours before they produce the required output. For example, you might want the computer to sort the names of all the people in Canada into alphabetical order, and then produce a list. You may become worried, after 30 minutes have elapsed with no output, that the program is in an infinite loop. In such circumstances, you can interrupt the program with the STOP key and "browse" around looking at partial results with BASIC immediate mode statements. If you are satisfied, you can use the command

```
CONTINUE
```

to resume execution at the beginning of the line where the program was interrupted. Unlike the RUN command, CONTINUE does not re-initialize the program but rather resumes execution where the program was interrupted with data and statement-control information preserved.

The CONTINUE command can also be used to resume executing programs which were interrupted by BASIC when it recognized an error. The error might be corrected by editing the program, or by altering data values with immediate mode statements.

If you wish to resume program execution at a different point than where it was interrupted, the primitive GO TO statement may be used in immediate mode to accomplish this. For example,

```
GO TO 50
```

will resume execution, with program state preserved as with CONTINUE, at line 50 of the program.

## 6.4 Monitoring the Operation of a Program

"Browsing" at program data does not always present enough clues to solve a problem. The command

STEP

can be used to resume an interrupted program, but execute one line at a time. Each program line is displayed before it is executed. After execution of each line, the computer waits until the RETURN key is pressed. If the STOP key is pressed instead of RETURN, execution is interrupted and microBASIC returns to Ready status. Note that commands can only be entered when microBASIC is in Ready status; commands are not recognized in STEP mode.

Another form of this command,

STEP line number

can be used to resume execution in STEP mode at a point other than where the program was interrupted. In fact, this form of the command allows STEP mode execution of a program that was not in an interrupted state.

Another method of monitoring execution is to insert PAUSE statements at selected locations within the program. The program is interrupted when a PAUSE statement is executed. This technique is more precise than using the STOP key with respect to where the program is interrupted. In many cases, strategic use of PAUSE statements can be more effective than STEPping through large programs one line at a time.

When a program is CONTINUEd after interruption by a PAUSE statement, execution resumes at the line after the PAUSE.

## 6.5 Testing the Program in Small Parts

As you might expect, the difficulty of debugging programs increases as they become larger and more complex. Programs which have been written in a modular fashion are typically easier to debug since different modules often can be tested independently.

In Waterloo microBASIC, this can be done by organizing major functional operations of the program into *procedures* or *functions*. These can be *called* (executed) with immediate mode statements to test them. For example, typing

```
call Display_menu
```

will immediately invoke execution of the procedure named 'Display_menu'.

## 6.6 Editing An Interrupted Program

When a program is interrupted the microBASIC system remembers which line to execute when resuming with CONTINUE or STEP. In the case of an error interruption, this line is that in which the error occurred. When the STOP key is used to interrupt execution, the line at which to resume is that which was about to be executed, that is, the line which is displayed. The line following a PAUSE statement is the point at which execution resumes after a PAUSE interruption.

If the program is edited after being interrupted, the position of resumption can be affected. If the line at which to resume is replaced, the new line becomes the resumption point. If it is deleted, the line with the next higher number becomes the resumption point. When the interrupt was the result of a PAUSE statement and a line is inserted between the PAUSE and the resumption point, the inserted line becomes the new resumption point.

Editing can cause problems with return to outstanding procedure, function and GOSUB calls. If the line which executed the call is deleted, changed or replaced, the program will be interrupted with an error when the return is executed.

# Chapter 7

# File Management

Some devices, such as disks, provide external storage on which files may be kept. Copies of programs may be stored in these files as described in earlier chapters. In addition, files may be created and accessed by BASIC programs for the purpose of storing, retrieving and updating data. A file might contain data such as marks for the students in a particular class. Data in files is retrieved with *input* operations and stored or updated with *output* operations. Devices which provide file storage are referred to as *file-oriented devices*.

Sooner or later, every user encounters the finite storage capacity limitations of the file storage device being used. The need for file management becomes painfully obvious when there is no space left to store important data or a large program that has been arduously typed and debugged. The DIRECTORY command provides a means of examining what files are stored on a device, how much space they occupy and how much space is available. Other commands allow files to be scratched in order to free space, and filenames to be changed. These commands are described in the following sections of this chapter.

## 7.1 DIRECTORY Command

The command

DIRECTORY 'filename'

displays a list of the files which are stored on the device or file-area specified. The format of the list produced depends upon the type of device or file-area specified. Usually, the amount of space occupied by each file and the amount of space available is indicated.

In the case of the Commodore disk, the directory listing is in column format. The first column indicates the number of 256-character blocks occupied by each file. The second column contains the file names. The third column identifies the file type. Copies of programs created with the STORE command have a file type of PRG. Ordinary data files and program copies created with the SAVE command have file type SEQ. Relative files have file type of REL. The last line of the directory listing indicates the number of blocks remaining unused.

Examples follow which illustrate use of the directory command with the Commodore disk.

| *command* | *meaning* |
|---|---|
| directory | list files on drive 0 of disk unit 8 |
| directory 'disk' | same as above |
| directory 'disk/1' | list files on drive 1 of disk unit 8 |
| directory 'disk9/1' | list files on drive 1 of disk unit 9 |

## 7.2 SCRATCH Command

The command

SCRATCH 'filename'

erases the file specified. When a Commodore disk file is SCRATCHed, its space is

made available for use with other files. Actually, SCRATCH is a BASIC programming language statement. It is used as a command with immediate-mode execution, but can be used in a program.

```
scratch 'charlie'
scratch 'disk8/0.charlie'
```

The above examples are equivalent. Both erase the file named "charlie" on drive 0 of disk unit 8.

## 7.3 RENAME Command

The command

RENAME 'filename' TO 'name'

changes the name of the file specified by "filename" to "name". This command does not physically move the file and consequently the device parameters of the filename cannot be changed. The entire string "name" is used as the name of the file within the device; this string should not contain a device specification such as disk or host. For example, the command

```
rename 'disk/1.customer' to 'oldcustomer'
```

changes the name of file "customer" on drive 1 of disk unit 8 to "oldcustomer".

The following example illustrates the confusion that can result from improperly including a device specification as part of the new name.

```
rename 'disk/1.abc' to 'disk/1.xyz'
```

The result is that file "abc" now has the name "disk/1.xyz"; that is, it would be referenced as disk/1.disk/1.xyz.

Like SCRATCH, RENAME is actually a programming language statement. It is used as a command with immediate-mode execution, but can also be used in a program.

## 7.4 MOUNT Command

The Commodore disk unit maintains information concerning the free space of each flexible disk in its drives. This information is kept in a table called the Block Availability Map (BAM). The BAM keeps track of how many blocks of space are available and where they are located. When a different disk is placed in a drive, a new BAM must be read with information for the new disk to avoid allocation of data blocks over existing files. Newer models of Commodore disks such as the 8050 automatically detect when a disk is placed in a drive and automatically read a new BAM. Older models, however, do not detect a change of disk and the MOUNT command must be used to instruct the disk unit to read a BAM. The MOUNT command is specified as follows:

```
MOUNT 'filename'
```

where the filename indicates the disk unit and drive number. Examples follow which illustrate the MOUNT command.

```
MOUNT 'disk'
MOUNT 'disk8/0'
```

The above commands are equivalent. Both cause a new BAM to be read for the disk in drive 0 of disk unit 8.

Note that the MOUNT command corresponds to the INITIALIZE command described in the Commodore disk manuals. Since the term "initialize" has the connotation of erasing or reformatting a disk on many computer systems, Waterloo microBASIC uses the command MOUNT instead.

Like SCRATCH and RENAME, MOUNT is actually a programming language statement. It is used as a command with immediate-mode execution, but can also be used in a program.

## 7.5 TYPE Command

The command

TYPE 'filename'

displays the contents of the specified file. If the file contains more records than can be displayed on the screen, the TYPE command pauses each time a screen is filled. Displaying of records continues if just the RETURN key is pressed. If another command is entered while TYPE is paused, TYPE terminates and the new command is processed.

## 7.6 SETUP Command

The command

SETUP

displays the SETUP menu to allow setting of appropriate characteristics for serial and host communications. Refer to the System Overview manual for further information concerning SETUP and serial/host communications.

# Chapter 8

# Using the General Editor

In addition to the editing capabilities provided by the microBASIC system, it is possible to make use of the general text editor. The general editor provides some powerful text manipulation facilities. For example, it is possible to change all occurrences of a name to another name in an entire program using only one command.

The command

EDIT

invokes this editor from microBASIC. The current program in the workspace can then be modified using the facilities of the general editor. The capabilities of the general editor are documented in the *Waterloo System Overview* manual. The editor's BYE command can be used to return to BASIC level.

**Warnings:**

1. When the editor is used, certain information about the current program state is lost. Because of this, a program is initialized, as with the RUN command, upon returning to microBASIC. Consequently, an interrupted program could not be modified with EDIT and have its execution resumed with the

CONTINUE command.

2. One shortcoming regarding use of the general editor for a BASIC program is that it does not recognize line numbers. Should you enter a line at a point in the program that is inconsistent with its line number, the editor will not diagnose this or reposition the line for you. Similarly, lines can be added without line numbers. These problems are noted by microBASIC when you attempt to execute the program.

   The BASIC-level RENUMBER command can be used to assign line numbers based upon the actual position of the line as entered with the general editor. Thus lines can be added to the program with the general editor without regard to line numbers and assigned numbers with RENUMBER.

3. The general editor, invoked from microBASIC, should not be used to edit data files. Used from microBASIC, the editor always provides space for line numbers since it is intended for BASIC programs. This can cause extra spaces to be inserted automatically at the front of data file records. To edit data files, leave microBASIC using the BYE command and select the general editor for stand-alone use.

# Waterloo microBASIC

# Programming Language Reference Guide

# Waterloo Computing Systems Newsletter

The software described in this manual was implemented by Waterloo Computing Systems Limited. From time-to-time enhancements to this system or completely new systems will become available.

A newsletter is published periodically to inform users of recent developments in Waterloo software. This publication is the most direct means of communicating up-to-date information to the various user. Details regarding subscriptions to this newsletter may be obtained by writing:

**Waterloo Computing Systems Newsletter**
**Box 943,**
**Waterloo, Ontario, Canada**
**N2J 4C3**

# Chapter 1

# A Program

## 1.1 Introduction

A *program* consists of a set of instructions which define actions to be performed by the computer. These instructions must be expressed in a language which the computer can recognize and understand. BASIC is a language which is designed to be a straightforward means of expressing instructions to the computer. A BASIC program is composed of *statements*: a statement which instructs the computer to perform an action is termed *executable*; one which defines characteristics of the program is termed *declarative*; and one which provides descriptive information to a person examining the program is termed a *comment*. Typically, each BASIC statement appears on a separate line. The order of lines in the program is determined by a line number which appears on the left side of each line. Consider the following example:

```
10 ! Simple Example
20 ! Print the number 5, its square and cube
30 print 5
40 print 5 * 5
50 print 5 * 5 * 5
60 stop
70 end
```

Lines 10 and 20 contain comments describing the program. Lines 30-50 contain executable statements which cause the numbers 5, 25 and 125 to be printed on the computer's terminal display. Note that the numbers 25 and 125 are calculated in an algebraic-like formula, or *expression*, wherein the asterisk (*) represents multiplication. The STOP statement in line 60 is executable and causes the program to terminate. The END statement in line 70 serves as a declarative, marking the end of the program.

The actions specified by a program's statements are performed when the program is *executed* or *run*. Refer to the Waterloo microBASIC Command Language portion of this manual for information concerning entering and running programs. The following output will be printed on the computer's display when this program is run:

```
Executing...
 5
 25
 125
Ready
```

The first message indicates that program execution has begun. The last message signals that the program has terminated and the computer is ready to accept commands.

## 1.2 Some Rules About Programs

### 1.2.1 Line Numbers

Line numbers must be specified for each line in a BASIC program. These numbers must be integer values in the range 1 through 65529. Leading zeroes are ignored in line numbers; i.e., 100, 0100 and 00100 are all treated as the same line number.

### 1.2.2 Keywords

Certain words, known as keywords, have a special meaning to BASIC. These include (i) words which define statement types such as STOP, (ii) special words used in statements such as TO, and (iii) intrinsic function names such as VALUE$. A complete list of Waterloo microBASIC keywords is given in an appendix of this manual. Keywords cannot be used as names of variables, arrays, functions or procedures.

Keywords can be entered in upper or lower case or in a combination of both; that is, Print, print and PRINT all represent the same keyword. Regardless of which way keywords are entered, they will always appear in lower case when the program is listed because of the manner in which the program is encoded as it is entered.

### 1.2.3 Use of Spaces

Spaces should be used in statements to improve readability of the program. Spaces cannot occur within line numbers, keywords, multi-character operators, or names of variables, arrays, functions or procedures. Spaces are required to separate keywords from names, numeric constants and other keywords.

### 1.2.4 Comments

Comments may be placed at the end of most lines. A comment starts with an exclamation mark, '!'. This indicates that the remainder of the line contains documentation that is to be ignored by the BASIC system. An entire line can be made a comment by placing an exclamation mark at its start.

In addition, comments may be entered into a program with the REM (remark) statement. When a statement starts with the keyword REM, the remainder of that line is treated as a comment.

### 1.2.5 Null Lines

It is permissible to enter a line containing only a line number and spaces or a comment. Such a line which contains no statements is called a null line. Null lines may be used to enhance the readability of programs.

### 1.2.6 Multiple Statements Per Line

More than one statement can be entered on one line, if sufficient space exists, by separating the statements with a colon, ':', character. This practice is generally considered to be a poor programming technique since it tends to render programs less readable and more difficult to modify.

Several statement types cannot be entered this way and must appear as the only statement on a line. These include DATA, DEF, FNEND, FOR, NEXT, IF, ELSE, ELSEIF, ENDIF, WHILE, LOOP, ENDLOOP, UNTIL, GUESS, ADMIT, ENDGUESS, PROC and ENDPROC statements.

Users of multiple statements per line should consider with care the ramifications with respect to error-handling facilities described in a later chapter. In general, it is recommended that only one statement be entered per line.

### 1.2.7 Multiple-Line Statements

A statement can be entered using multiple lines. A line containing a statement, which is to be continued on the following line, must contain an ampersand, '&', character as its last non-blank character. The following line must contain an ampersand as the first non-blank character after the line number.

```
100 PRINT #4,   SITE_REVENUE,       ! SITE SUMMARY&
110 &           REGION_REVENUE,     ! REGION SUMMARY&
120 &           COUNTRY_REVENUE,    ! COUNTRY SUMMARY
```

The above example illustrates continuation of a statement over three lines. Note that comments can appear, as illustrated, on each line of a continued statement.

DATA statements cannot be continued. Program entities, such as keywords, constants, names and operators cannot span continued lines.

### 1.2.8 END Statement

The strict rules of BASIC require that every program have an END statement as its last statement. A program will, however, execute properly if the END statement is missing. It is an error to place an END statement anywhere in the program except as the last statement.

# Chapter 2

# Types of Data

All programs use information or *data*. There are two classes of data in a BASIC program. One class is *numeric* data, that is information consisting of numbers. The other class is *string* data, that is sequences of alphabetic and other characters. These types of data can be represented in various ways. This chapter describes the manner in which data can be represented with microBASIC.

## 2.1 Numeric Constants and Variables

A numeric constant is a numeric value, such as 12 or 3.5, which does not change during the execution of a program. A numeric variable is a name, such as Total, which represents one specific value at any given instant, but can be assigned different values throughout the execution of a program. Variables are used in a manner similar to the manner in which symbols or names are used in scientific and business equations or formulae. In this section, the rules for numeric constants and variables are given.

### 2.1.1 Numeric Variables

A numeric variable is written as a sequence of alphabetic characters, digits and underscore ("_") characters. The variable must start with an alphabetic character and can be up to 31 characters in length. It is advisable to use names which are descriptive of the usage of the variable. Examples of numeric variables follow:

```
Sum_of_Squares
Page_Number
LINE_COUNTER
Student_average
```

As illustrated, both upper and lower case alphabetic characters may be used in a variable name. Note that upper and lower case letters are *not* treated as if they are equivalent; that is, TOTAL represents a different variable than Total. Keywords cannot be used as variable names.

At any particular time there is one value associated with a variable. When a program is run, each variable is automatically assigned a value of zero. Other values may be assigned to the variable by execution of BASIC statements such as assignment statements. Programs should not be written with the assumption that variables are initially assigned zero values. It is good programming practice to assign explicitly an initial value to each variable in a program before the variable is first used.

### 2.1.2 Numeric Constants

Numeric constants are entered in BASIC programs in a number of formats, as illustrated below:

```
 8152
 400.37
 .36945
-63
-.00395
 894871.
```

Each constant can consist of a sign (+ or -), followed by a sequence of integer digits, a decimal point and a sequence of fractional digits. These elements are combined to represent a numeric value in the manner that people normally write numbers except that commas cannot be used in a numeric constant:

```
7,194,632.79
7,432
6.479.321,437
```

The preceding specifications are all incorrect and will cause errors.

Numeric constants may also be entered in scientific notation. In this notation, a constant as described above is followed by an 'E' or 'e' and a second integer value.

| *Scientific Notation* | *Value* |
|---|---|
| 7.36E2 | 736 |
| 21.437e-6 | .000021437 |
| -.098E+4 | -980 |

In the preceding example, the first column illustrates numeric constants in scientific notation and the second column gives the corresponding values. The actual value to be represented is determined by taking the number preceding the 'E' and multiplying it by ten raised to the power of the integer following the 'E'. The number preceding the 'E' is called the mantissa and the integer following the 'E' is called the exponent. Scientific notation is commonly used when the magnitude of numbers is either very large or very small.

There are no rules in BASIC which specify where scientific notation should or should not be used. A programmer should use whichever format makes the program more easily understood by another person.

### 2.1.3 Internal Representation of Numeric Data

Numeric variable and constant values are represented internally in a form called floating-point. Floating-point representation differs from one type of computer to another. Every computer has a maximum absolute numeric value that can be represented. This is sometimes called *machine infinity*. A special function, named INF, gives this maximum value for the computer being used. To display this value, enter the following statement:

```
print inf
```

Likewise, every computer has a minimum absolute non-zero numeric value that can be represented. This value, called *machine epsilon,* can be similarly displayed using the special function, EPS.

When values with an absolute magnitude greater than machine infinity are calculated, an *overflow* condition results and the maximum value is substituted. Similarly, when values with an absolute magnitude smaller than machine epsilon are calculated, an *underflow* condition results and zero is substituted. Refer to the chapter entitled ERROR HANDLING for further information regarding these error conditions.

Floating-point representation of numbers is approximate for many values; however, the amount of error is normally quite small. When a numeric constant is converted to this internal format, it will be correct to a number of significant digits that is dependent upon the type of computer used (9 digits for your personal computer and 16 digits for an IBM Series/1 minicomputer or IBM 370 computer). The evaluation of an expression can compound this error causing it to become greater than when the initial conversion occurred. It is beyond the scope of this manual to discuss this phenomenon thoroughly. In most cases the amount of error is too small to impact the computations in a program. Error-free representation of numbers and computational results can be attained using integer variables, introduced in the following section, for the class of values that can be represented as integer variables.

### 2.1.4 Integer Variables

A special type of variable can be used to represent integer values in the range -32768 to +32767. This seemingly strange value range results from the fact that integer variable values are represented internally in binary (base 2) integer format. Sixteen bits, each defining a binary digit, 0 or 1, are used to represent integer variable values. The most significant bit (conceptually, the farthest-left bit) indicates the sign, with zero representing plus, one representing minus. The remaining 15 bits can represent 2 raised to the power 15 unique positive values (0 through 32767) and equal number of negative values (-32768 through -1).

Integer variable names are similar to numeric variable names with a percent symbol, '%', appended to the end of the name. That is, an integer variable name must start with an alphabetic character and consists of up to 31 alphabetic characters, digits and underscore characters, '_', followed by a percent symbol. Examples of integer variables follow:

```
Page_Number%
COUNT%
Part_number%
```

Integer variables are automatically assigned a value of zero each time a program is run.

## 2.2 String Constants and Variables

A string value is a sequence of characters. A string constant is a string value that does not change during the execution of a program. A string constant is enclosed by a pair of quotation marks; single (') or double (") quotation characters can be used. The same character must be used to delimit the start and end of a particular string constant. If one type of quotation mark is a character in the string, the other type must be used to enclose the string. Examples of string constants follow:

```
'This is a string.'
"SALES TOTAL"
"There isn't enough"
'Digits: 0 1 2 3 4 5 6 7 8 9'
""
```

A string containing no characters, such as the last example, is called a *null string* and has length zero. The maximum size of a string value is usually restricted by the size of the computer's memory available for storing the characters; however, when large computers are used, a string value is limited to a maximum of 65535 characters.

A string variable is a name which represents one specific string value at any given instant, but can be assigned different values during the execution of a program. A string variable name must start with an alphabetic character and has up to 31 alphabetic characters, digits and underscore characters, followed by a dollar symbol. Keywords cannot be used as variable names. The following are examples of string variable names:

```
Name$
STREET$
City_State$
```

String variables are automatically assigned the null string as an initial value each time a program is run.

## 2.3 Matrices

A *matrix* is used to represent a list or table of values. Another name for a matrix is an *array*. Like simple variables, which only represent a single value, matrices can be numeric, integer or string. Matrix names start with an alphabetic character and can contain up to 31 alphabetic characters, digits and underscore characters. Integer matrix names end with a percent symbol '%'. String matrix names end with a dollar symbol '$'.

Matrix values can be referenced individually or collectively. All elements of matrices are automatically set to an initial value each time a program is run: the initial value for numeric and integer matrix elements is zero; the initial value for string matrix elements is the null string.

Matrices can be various shapes and sizes according to their dimensions. A matrix of one dimension is a list or vector of values, the number of which are defined by the matrix dimension. A two-dimensional matrix is a table of values, the number of rows being determined by the first dimension and the number of columns being determined by the second dimension. A three-dimensional matrix can be considered as a number of tables, the number being determined by the third dimension. Matrices can have four or more dimensions, but matrices with more than three dimensions become difficult to conceptualize and are seldom used. The number and size of dimensions that can be specified is limited practically by the amount of memory available for the matrix values.

The number and size of dimensions for a particular matrix are defined the first time a reference to the matrix is encountered during execution of a program. Thereafter, all references to elements of that matrix must be consistent with respect to the number of dimensions and the bounds of each dimension as defined.

### 2.3.1 DIM Statement

Syntax: DIM name(dimensions), name(dimensions) ...

where

| | |
|---|---|
| *name* | is a matrix name |

and

| | |
|---|---|
| *dimensions* | is a list of numeric values, separated by commas |

The DIM (dimension) statement is used to define the number and size of the dimensions for one or more matrices. The numeric values which specify the dimensions can be numeric constants, variables or expressions which yield numeric results when evaluated. The numeric values are rounded to the nearest integer to define the size of a dimension. The number of dimensions for a matrix corresponds to the number of values, separated by commas, between the parentheses.

```
40  DIM  PRICE(100)
```

The above example shows a DIM statement which defines a numeric matrix named PRICE with one dimension and 101 values or elements. The individual elements of this matrix can be referenced as PRICE(0), PRICE(1), PRICE(2), ..., PRICE(100).

The following example defines a string matrix NAME$ and an integer matrix AGE%, each having 21 elements.

```
100  DIM  NAME$(20), AGE%(20)
```

The number of elements in a multi-dimensional matrix is calculated as the product of each dimension incremented by 1. Thus, the matrix defined below to have 11 rows and 4 columns, has a total of 44 elements.

```
80  DIM  SCORE(10,3)
```

The number of elements required in a particular matrix may differ each time a program is run. In this type of situation, variable dimensions are useful. The following matrix definition might be used to keep track of the scores for a number of contestants each competing in a number of events.

```
80  DIM  SCORE(CONTESTANTS, EVENTS)
```

*Rules*:

(1) The dimensions of each matrix can only be defined once in each program.

(2) If a DIM statement is used to define the dimensions of a matrix, the DIM statement must be executed prior to any other references to that matrix.

(3) When a matrix is referenced in a program without the dimensions having been defined by a DIM statement, dimensions are defined by default to a value of 10. Individual elements of a matrix are referenced with subscripts as described in the next section of this manual. If an individual element of a matrix is referenced before its dimensions are defined, the number of dimensions is defined to correspond to the number of subscripts used. Some statements, such as MAT assignment, support operations involving entire matrices. When a matrix as a whole is referenced in this manner, without its dimensions having been established, an error results.

(4) The value for each dimension must be a non-negative number.

### 2.3.2 Matrix Subscripts

Some BASIC statements support operations involving entire matrices. Frequently, however, it is useful to reference individual values or elements of a matrix in the same context that simple variables are used in a program. This is done using subscripts. For example, element 5 of a one-dimensional matrix PRICE could be referenced as PRICE(5). In this example the subscript is 5.

Subscripts are enclosed in parentheses following the matrix name. If the matrix has more than one dimension, a list of subscripts is specified separated by commas. The number of subscripts must equal the number of dimensions defined for the matrix.

Subscript values must be numeric and can be constants, variables or expressions which yield numeric results when evaluated. Each subscript value is rounded to the nearest integer and must not exceed the corresponding dimension value or be less than zero.

An element in the two-dimensional matrix SCORE might be referenced as SCORE(ATHLETE, SPORT). The row and column of the element are established by the values of variables ATHLETE and SPORT, respectively.

### 2.3.3 OPTION BASE 1

Mathematicians and scientists often express formulae in terms of vectors which have an element 0 or arrays with 0-th rows and columns. Many other people prefer to number elements of a list starting with 1. The statement

```
OPTION BASE 1
```

specifies that *all* matrices in a program have 1 as the first element in each dimension. If this statement is used, it must be executed before any matrices are referenced in a program.

```
30 OPTION BASE 1
40 DIM PRICE(100)
50 DIM SALESMAN$(5)
```

The above example defines a numeric matrix PRICE having 100 elements, PRICE(1), PRICE(2),...,PRICE(100), and a string matrix SALESMAN$ having 5 elements SALESMAN$(1), SALESMAN$(2),..., SALESMAN$(5).

# Chapter 3

# Expressions

An expression defines the computation of a new value from other values using *operators* such as + which specifies addition. Constants, variables, matrix elements and function references can be used as values in an expression. The simplest form of an expression involves no computation and is simply a value with no operators. Examples of simple expressions follows:

```
1234.567
'characters'
Price
```

Evaluation of expressions which have only one operator is straightforward and intuitive. For example, the value of the expression

```
Price + Tax
```

is computed by adding the values of two numeric variables, Price and Tax.

The rules for evaluating expressions with a number of operators are more complex. The order in which operations are performed is determined not only by the order in which they appear (left to right), but also by each operator's *priority*.

For example,

```
1 + 2 + 4
```

is evaluated by adding 1 and 2 giving 3, then adding 4 giving 7; whereas

```
1 + 2 * 4
```

is evaluated by multiplying 2 * 4 giving 8 and adding 1 to give 9.

## 3.1 Priority of Operators

Operations of higher priority are performed in an expression before operations of lower priority. Operations are performed in a left to right order in cases of equal priority.

| *Priority* | *Operation* | *Example* |
|---|---|---|
| 11 | matrix subscripting | table(i,j) |
| | function reference | fna(x,y) |
| 10 | substring | name$( start : end ) |
| 9 | enclosed in parentheses | 3 * ( i+1 ) |
| 8 | exponentiation ↑,** | value ↑ 3 |
| 7 | unary plus + | +5 |
| | unary minus - | -2 |
| 6 | multiplication * | quantity * unit_cost |
| | division / | gallons / 4 |
| 5 | addition + | total + amount |
| | subtraction - | total - credit |
| | concatenation + | province$ + ", Canada" |
| 4 | relational operators | |
| | =,>,<,<>,>=,<= | total >= 50 |
| 3 | logical NOT | NOT age% > 30 |
| 2 | logical AND | price > 10 AND quantity < 3 |
| 1 | logical OR | errors > 5 OR average < 50 |

These operations are described in the next several sections of this chapter.

## 3.2 Matrix Subscripting

As introduced in the previous chapter an individual element of a matrix can be specified by subscripting. Each subscript must be an expression which yields a numeric result. Each subscript value is rounded to the nearest integer and then used as an index to identify the individual element.

## 3.3 Function Reference

BASIC provides a number of intrinsic functions which yield a value when referenced in an expression. For example, the LEN function yields a numeric value corresponding to the length of the string which is specified as a parameter for this function. The value of LEN('abc') is 3, that is, the number of characters in parameter 'abc'. Some intrinsic functions return string values. These have names ending with a dollar symbol, $. A programmer can define functions as well which can be referenced in expressions in the same manner as intrinsic functions. Definition of functions is described in a later chapter.

Parameters in function references are expressions. Each parameter expression is evaluated and *passed* to the function which normally uses these values in computing a result.

## 3.4 Substring

A portion of a string value can be extracted using the substring operation. The string value to which the substring operation applies must be a string variable or a subscripted element of a string matrix.

```
var$( start : end )
matrix$( subscripts )( start : end )
```

The portion of the string to be extracted is specified by the position of its starting and ending characters, as two numeric valued expressions separated by a colon. Both start and end values are rounded to integers.

For example, if the value of string variable Letter$ is 'abcdef' the value of substring

```
Letter$(2 : 4)
```

is 'bcd'. If the end value exceeds the number of characters in the string, the last

character of the string is treated as the end. Thus, the value of

```
Letter$ (3 : 20)
```

is 'cdef'. Similarly, if the start value is less than 1, the first character of the string is treated as the start.

The null string results from any of the following situations: the original string is null; start value exceeds end value; start value exceeds the length of the string; end value is less than 1.

## 3.5 Parenthesized Operations

The normal priority of operations can be pre-empted by enclosing operations in parentheses, with the exception of subscripting, function references and substring operations. For example, the expression

```
1 + 2 * 4
```

is evaluated by multiplying 2 and 4, then adding 1 giving 9; whereas the expression

```
(1 + 2) * 4
```

is evaluated by adding 1 and 2, then multiplying by 4 giving 12. More generally, subexpressions can be enclosed by pairs of parentheses. Parenthesized operations within a parenthesized subexpression are said to be *nested*. For example, the expression

```
10 * (48 - (45-3))
```

is evaluated by subtracting 3 from 45 giving 42, subtracting this from 48 giving 6 and multiplying by 10 to give 60.

## 3.6 Exponentiation

Exponentiation computes a numeric value raised to a numeric power. This is expressed in the form

```
value ↑ power  or  value ** power
```

For example, 2 ↑ 3 is 8. Fractional and negative powers are permitted. For example,

4 ↑ .5 is the square root of 4, namely 2. Similarly, 8 ↑ (-1) is the reciprocal of 8, namely .125 or one-eighth.

A number of special cases exist and are shown in the table following.

| *Special Case* | *Result* | *Condition* |
|---|---|---|
| n ↑ 0 | 1 | for n < 0, n = 0, n > 0 |
| 0 ↑ n | 0 | for n > 0, n < 0 |
| n ↑ m | error | for n < 0 if m not an integer value |

### 3.7 Unary Plus

Unary plus,

+ value

produces a numeric result identical to the numeric value operated upon.

### 3.8 Unary Minus

Unary minus,

− value

produces a numeric result which is the numeric value operated upon with its sign reversed.

### 3.9 Multiplication

Multiplication,

value * value

gives a numeric result which is the product of two numeric values.

### 3.10 Division

Division,

value / value

gives a numeric result which is the quotient obtained from dividing the first numeric value by the second. Division by zero is an error.

This error condition can be handled by the ON ZDIV statement described in the chapter concerning Error Handling. If no ON ZDIV handling of this error is specified, a message is displayed, machine infinity is substituted as the result and the program continues.

### 3.11 Addition

The result of addition,

value + value

is the numeric sum of two numeric values.

### 3.12 Subtraction

The result of subtraction,

value - value

is the numeric difference of two numeric values, subtracting the second from the first.

### 3.13 Concatenation

The result of concatenation,

value + value

is a string consisting of the characters of the first string value followed by the characters of the second string value. For example,

'John' + 'son'

produces the string

'Johnson'

### 3.14 Comparison (Relational Operations)

Values can be compared using the relational operators listed below:

| *relation* | *meaning* |
|---|---|
| value = value | equals |
| value > value | greater than |
| value < value | less than |
| value <> value | not equal to |
| value >= value | greater than or equal |
| value <= value | less than or equal |

Note that the last three operators are composed of two characters each. These operators can also be entered as ><, => and =<; however, since the operators are encoded when a statement is entered, the operators always appear as shown in the table when the program is LISTed.

The result of a comparison is a numeric value of 1 if the relation is true, and 0 if the relation is false. String values must be compared with string values and numeric with numeric. It is an error to compare numeric with string values.

Tests for equality can produce surprising results when non-integer numeric values are compared. This is caused by the approximation of values in the internal representation of numbers and the rounding errors that can accumulate when arithmetic is performed using these approximate values. Refer to the description of numeric internal representation in the chapter concerning types of data.

Characters are represented internally in computers by numeric codes. Each character is assigned a unique code in the *character set* used by the computer system. Characters are compared according to their relative position in the character set, that is, according to their numeric code values. Strings are compared character by character, from left to right. If the strings differ in length, they are compared using the shorter length; if they match, the longer string is considered greater than the other string.

### 3.15 Logical NOT

Logical NOT,

NOT value

operates on a numeric value, giving numeric result 1 (true) if the value is 0, and giving result 0 (false) if the value is non-zero.

When the value is an integer variable, each bit (binary digit) of the value is operated upon independently, resulting in an integer value composed of individual bit results. The reader should have a thorough understanding of binary representation and operations when using this feature.

### 3.16 Logical AND

Logical AND,

value AND value

operates on two numeric values, giving numeric result 1 (true) if both values are non-zero, and giving result 0 (false) if either value is zero.

When both values are integer variables, corresponding bits (binary digits) of the values are operated upon independently, resulting in an integer value composed of individual bit results. The reader should have a thorough understanding of binary representation and operations when using this feature.

### 3.17 Logical OR

Logical OR,

value OR value

operates on two numeric values, giving numeric result 1 (true) if either value is non-zero, and giving result 0 (false) if both values are zero.

When both values are integer variables, corresponding bits (binary digits) of the values are operated upon independently, resulting in an integer value composed of individual bit results. The reader should have a thorough understanding of binary representation and operations when using this feature.

### 3.18 Integer Operations

Arithmetic expressions involving only integer variables are evaluated using integer computations and produce integer results. For example, the operation

```
three% / two%
```

where three% has value 3 and two% has value 2, produces a result of 1 rather than 1.5.

Any arithmetic operation involving a numeric constant or a non-integer numeric variable is evaluated using floating-point computations. If an integer variable is used in a operation with another type of numeric value, the integer value is converted automatically to floating-point representation before the operation result is computed.

Manipulation of binary values, bits, can be accomplished using integer variables with AND, OR and NOT operations.

# Chapter 4

# LET Statement

## 4.1 Assignment in General

The LET statement is used to assign a value to a variable, a matrix element, a function result, or a substring of either a string variable or matrix element. The value to be assigned is specified as an expression following the equals symbol "=". The receiving item precedes the equals symbol. The keyword, LET, can optionally precede the receiving item. The following examples are equivalent.

```
let x = 5 * y
x = 5 * y
```

In practice, the keyword LET is seldom used.

Only numeric expression results can be assigned to numeric or integer receiving items. Similarly, only string expression results can be assigned to string receiving items. When a non-integer value is assigned to an integer receiving item, the fractional portion of the value is truncated (discarded).

## 4.2 Matrix Element Assignment

Matrix elements are assigned a value by specifying a matrix subscript preceding the equals symbol. Matrix subscripts are described in an earlier chapter concerning types of data. Examples follow,

```
table(i,j) = 10 * i + j
name$(student_number) = 'charlie'
```

## 4.3 Function Result Assignment

Function results are assigned by specifying the function name as the receiving item. It is valid to assign a value to a function name only if it is the name of the currently active function. Examples of assignment to a function name follow.

```
fna  = x * y
fna% = result%
fna$ = 'abc'
```

Refer to the chapter entitled "Functions" for further information.

## 4.4 Substring Assignment

Substring receiving items are specified in a manner similar to substrings in expressions. The characters defined by the substring are replaced in the receiving string by the resulting value of the expression. The new string is formed by concatenating three component strings:

(i) the characters, if any, preceding the substring in the original string;

(ii) the result of the expression following the "=" symbol;

(iii) the characters, if any, following the substring in the original string.

For example, if variable a$ had the value 'abcdef', the statement

```
a$(3:4) = 'CD'
```

causes a$ to have the value 'abCDef'; that is, characters "cd" in the original string are replaced by "CD".

The string value assigned need not have the same length as the receiving substring. For example, the statement

```
a$(5:5) = 'EEE'
```

now causes the value of a$ to be 'abCDEEEf' ("e" is replaced with "EEE"). Similarly, the statement

```
a$(5:7) = 'E'
```

now results in a$ having the value 'abCDEf' ("EEE" is replaced by "E").

As described in the chapter concerning expressions, some substring specifications define a null string. When such substrings are assigned a value, the effect is that the value assigned is "inserted" in the receiving string in front of the character defined by the substring start position. The following examples illustrate insertion in the middle, at the front, and at the end of a string:

```
a$(4:3) = '0'
```

gives a$ the value 'abC0DEf'; then

```
a$(0:0) = '0'
```

gives a$ the value '0abC0DEf'; and then

```
a$(99:0)= '0'
```

gives a$ the value '0abC0DEf0'.

The last example illustrates that regardless of how large the start value is specified, if it exceeds the length of the original string, the substring is treated as a null string adjacent to the end of the original string.

# Chapter 5

# Matrix Assignment

Assignment of values to individual matrix elements and use of individual matrix elements in expressions has been described in earlier chapters. A special set of matrix statements support assignment to an entire matrix including special expressions with operations involving entire matrices. In a particular matrix assignment statement, all matrices involved must have identical dimensions, with the exception of transposition and matrix by matrix multiplication.

All matrix assignment statements have the following form:

MAT matrix-name = matrix-expression

Each matrix expression can be one of the forms described in subsequent sections of this chapter. The different forms of matrix expressions *cannot* be combined in one statement to form more complex expressions.

## 5.1 Scalar to Matrix Assignment

Syntax: MAT matrix = (expression)

An expression which results in a scalar (single) value can be specified enclosed in parentheses. This value is assigned to each element of the matrix specified. Examples follow to illustrate this type of matrix assignment.

```
MAT QUANTITY = (1)
MAT A$ = ('a')
MAT X = (5 * p + q)
```

In each of these examples, a single value is derived from evaluating the parenthesized expression. This value is assigned to all elements of the matrix in each case.

Note that only a string value can be assigned to a string matrix and only a numeric value can be assigned to a numeric or integer matrix.

## 5.2 Special Matrix Constants

Syntax:
```
MAT numeric-matrix = ZER
MAT string-matrix  = NULL$
MAT numeric-matrix = IDN
```

Special matrix constants ZER, NULL$ and IDN can be assigned to matrices. These constants can only be used in matrix assignment statements. The first special constant ZER, when assigned to a numeric matrix causes all elements to have the value 0. The following two statements are equivalent:

```
MAT A = ZER
MAT A = (0)
```

When the matrix constant NULL$ is assigned to a string matrix, all elements of the matrix are assigned a null string. The following two statements are equivalent:

```
MAT A$ = NULL$
MAT A$ = ("")
```

Matrix constant IDN represents the identity matrix. This constant can only be assigned to a square numeric matrix, that is, a two-dimensional matrix with both dimensions being equal. The identity matrix has value 1 in all diagonal elements, that is, elements with equal row and column positions. All other elements of the identity

matrix have value 0. The following example shows assignment of the identity matrix to a 9-element square matrix.

```
10 DIM A(2,2)
20 MAT A= IDN
```

After execution of these statements, matrix A would represent values as illustrated below.

```
1   0   0
0   1   0
0   0   1
```

## 5.3 Matrix to Matrix Assignment

Syntax: MAT matrix = matrix

This type of matrix assignment copies the values of one matrix to the corresponding elements of another matrix. String matrices can only be assigned to string matrices. Numeric or integer matrices can be assigned to numeric or integer matrices. When appropriate, conversions between floating-point and integer internal representations are performed for each element. Examples follow:

```
MAT A = B
MAT X = I%
MAT I% = X
MAT S$ = T$
```

## 5.4 Addition, Multiplication of Scalar and Matrix

Syntax: MAT numeric-matrix = (expression) + numeric-matrix
MAT numeric-matrix = (expression) * numeric-matrix

A parenthesized expression which results in a scalar (single) numeric value can be added to, or multiplied by, each element of a numeric matrix individually. The arithmetic results are assigned to the corresponding elements of the numeric matrix specified preceding the equals symbol.

```
MAT Interest = (.18) * Capital
MAT Count% = (1) + Count%
MAT Canadian_funds = (1 + US_exchange) * US_funds
```

## 5.5 Addition, Subtraction of Two Matrices

Syntax:

```
MAT numeric-matrix = numeric-matrix + numeric-matrix
MAT numeric-matrix = numeric-matrix - numeric-matrix
```

Corresponding elements of two numeric matrices can be added or subtracted using this form of matrix assignment statement, and assigned to corresponding elements of the numeric matrix specified preceding the equals symbol. Examples follow.

```
MAT Total_Sales = New_Sales + Total_Sales
MAT Profit      = Price - Cost
MAT Inventory%  = Inventory% - Quantity_Sold%
```

## 5.6 Multiplication of Two Matrices

Syntax: MAT numeric-matrix = numeric-matrix * numeric-matrix

Multiplication of two numeric matrices, as mathematically defined, can be performed using this type of matrix assignment statement. The numeric matrix receiving the result *cannot* be one of the two matrices multiplied. Each of the three matrices must have two dimensions. The receiving matrix must have the same number of rows as the first matrix multiplicand, and the same number of columns as the second matrix multiplicand. The number of columns of the first matrix multiplicand must equal the number of rows of the second.

```
10 DIM A(p,q), B(p,r), C(r,q)
20 MAT A = B * C
```

The above statements conform to the rules regarding dimensions for matrix multiplication. The element of matrix A in row i (0<= i <=p) and column j (0<= j <=q) is calculated as follows.

```
A(i,j) = B(i,0)*C(0,j) + B(i,1)*C(1,j) + ... + B(i,r)*C(r,j)
```

## 5.7 Matrix Transposition

Syntax: MAT numeric-matrix = TRN(numeric-matrix)

The mathematical transpose of one numeric matrix can be assigned to another numeric matrix using this form of matrix assignment statement. Both matrices must have two dimensions and the number of rows in each must equal the number of columns in the other. The elements of each *row* of the receiving matrix are assigned the elements of the corresponding *column* of the other matrix.

```
10 DIM A(p,q), B(q,p)
10 MAT A = TRN(B)
```

The above statements conform to the rules regarding dimensions for matrix transposition. Each element A(i,j) is assigned the value of B(j,i), for values of i between 0 and p inclusive and values of j between 0 and q inclusive.

Note that the receiving matrix *cannot* be the same matrix that is transposed.

# Chapter 6

# Structured Control

Statements of a BASIC program are executed one at a time. Normally, statements are executed sequentially in order of ascending line number. This order of execution is satisfactory for very simple programs, however, most program applications require that different actions be chosen under different circumstances.

*Control statements* can be used to alter or control the order in which statements are executed. A set of control statements, described in this chapter, can be used to delimit groups of statements which are to be repeated or optionally executed. These *statement groups* and the statements which delimit them are known as *control structures*.

Several of these control statements are used to select or repeat a statement group conditionally based upon the value of an expression. Such expression results are numeric, but are treated as logical values *true* or *false;* any non-zero result is treated as true and a zero result is treated as false.

Primitive control statements exist which permit arbitrary transfer of control to any line of a program. Primitive control statements are described in a later chapter.

The group of statements delimited by control statements forming a structure can include other control structures. This is called *nesting* of control structures. Control

structures of the same or different types can be nested within a given structure. It is a common practice to indent statements within a structure, using leading spaces, to make programs more readable. Further indentation is used with each level of nesting to emphasize the structure of the program.

## 6.1 Repetition Structures

A structure which defines a group of statements that are to be repeated is often called a *loop*. Several means of defining loops exist. Different loop types are appropriate according to when information for control decisions is available.

### 6.1.1 FOR-NEXT Loops

Sometimes it can be determined before starting a loop, how many times the loop is to be repeated. The FOR-NEXT control structure provides a convenient means of controlling repetition of such loops.

```
Syntax:   FOR variable = expression TO expression STEP expression
            .
            . statement(s)
            .
          NEXT variable
```

The FOR-NEXT loop uses a variable as an index or control variable. When the FOR statement is encountered, the index variable is assigned the value of the expression following the equals symbol. The other expressions in the FOR statement are also evaluated at this time and are not re-evaluated during repetition.

The keyword STEP and its expression are optional and need not be specified. The STEP expression specifies the value by which the index variable is to be incremented or decremented after each repetition of the loop, that is, when the NEXT statement is encountered. If no STEP value is specified, an increment value of 1 is used.

Before each repetition of the loop, including the first, the index variable is compared with the value of the TO expression. If the index is past this value, the statements in the loop are not executed and control passes to the statement in the line following the NEXT statement. If the STEP value is negative, the loop is not repeated when the index is less than the TO value; otherwise, the loop is not repeated when the index exceeds the TO value.

Execution of a NEXT statement without first executing its corresponding FOR statement causes an error. Note that the NEXT statement *must* have a variable which matches that used as the index of the corresponding FOR statement. Either numeric or integer variables can be used as indices.

### 6.1.2 WHILE-ENDLOOP

```
Syntax:   WHILE  expression
            .
            .  statement(s)
            .
          ENDLOOP
```

The WHILE and ENDLOOP statements delimit a group of statements that is to be repeated. Before each execution of the statement group, including the first, the numeric expression in the WHILE statement is evaluated. If the value is true the statement group is executed; otherwise, control passes to the statement in the line following the ENDLOOP.

### 6.1.3 LOOP-UNTIL

```
Syntax:   LOOP
            .
            .  statement(s)
            .
          UNTIL  expression
```

The LOOP and UNTIL statements delimit a group of statements that is to be repeated. After each execution of the statement group, the numeric expression in the UNTIL statement is evaluated. If the value is true the statement group is not repeated and control passes to the statement in the following line.

### 6.1.4 WHILE-UNTIL

Syntax:
```
WHILE  expression
  .
  .  statement(s)
  .
UNTIL  expression
```

The WHILE and UNTIL statements delimit a group of statements to be repeated. Before each execution of the statement group, including the first, the numeric expression in the WHILE statement is evaluated. If the value is true the statement group is executed; otherwise, control passes to the statement in the line following the UNTIL.

After each execution of the statement group, the numeric expression in the UNTIL statement is evaluated. If the value is true control passes to the statement in the following line; otherwise, control passes to the WHILE statement where its conditional expression is evaluated again.

### 6.1.5 LOOP-ENDLOOP

Syntax:
```
LOOP
  .
  .  statement(s)
  .
ENDLOOP
```

The LOOP and ENDLOOP statements delimit a group of statements that is to be repeated. This structure defines an *infinite loop*, a loop that will be repeated endlessly unless the STOP key is pressed or the electrical power turned off. Many applications exist in which infinite loops are useful. Consider, for example, a program to control a digital clock.

This structure is also useful for non-infinite loops. Often information required to decide when to terminate a loop is not available at its beginning and is not appropriate to test at its end. These programming situations can be handled using a LOOP-ENDLOOP structure and an exit from the middle of its statement group using a QUIT statement. A QUIT can be conditionally executed by making it the object of an IF-THEN statement. The QUIT and the IF-THEN statements are described later in this chapter.

## 6.2 Choice, Selection

Control statements and structures are available which allow a program to choose if a single statement or statement group is to be executed. Additional control structures permit choice or selection of one statement group among several to be executed.

### 6.2.1 IF-THEN-statement

Syntax: IF expression THEN statement

The IF-THEN statement supports the choice of whether or not to execute the statement which follows the keyword THEN. The numeric expression is evaluated and if its value is true the statement following THEN is executed. This statement must be one which is meaningful by itself; for example, it could not be a structure delimiter such as LOOP or NEXT. The QUIT statement can be used with IF-THEN to conditionally exit from within an enclosing structure's statement group. The QUIT statement is described later in this chapter.

Another form of the IF-THEN statement is described in a later chapter concerning Primitive Control.

### 6.2.2 IF Structure

```
Syntax:  IF    expression
          .
          .  statement(s)
          .
         ELSEIF expression
          .  statement(s)
         ELSEIF expression
          .  statement(s)
         ELSE
          .  statement(s)
         ENDIF
```

This structure permits choice of one or none of possibly several different statement groups. The ELSEIF and ELSE statements are optional. If neither are specified, the structure appears as follows:

```
IF  expression
      .
      . statement(s)
      .
ENDIF
```

In this case, the numeric expression in the IF statement is evaluated and if its value is true the statement group is executed. If the value is false control passes to the statement in the line following the ENDIF.

One of two statement groups can be selected by using an ELSE statement in this structure as follows:

```
IF  expression
      .
      . statement(s)
ELSE
      .
      . statement(s)
ENDIF
```

In this case, a true value of the expression causes the statement group following the IF statement to be executed after which control is passed to the statement in the line following the ENDIF; otherwise, the statement group following the ELSE is executed.

Use of the ELSEIF statement permits one of multiple statement groups to be selected for execution. When ELSEIF is used, the IF expression is evaluated and if false the ELSEIF expressions are evaluated one at a time in order of appearance. The first IF or ELSEIF expression to be found true has its statement group executed and control is passed to the statement in the line following the ENDIF without evaluating any further ELSEIF expressions. If none of the expressions are found to be true, control passes to the line after the ENDIF unless an ELSE statement is part of the structure. If an ELSE is specified, its statement group is executed when all IF and ELSEIF expressions are found to be false.

Only one ELSE can appear in a single IF structure, however, an arbitrary number of ELSEIF statements can be used. An ELSE statement in an IF structure must follow any ELSEIF statements used in the same structure.

## 6.3 QUIT Statement

Syntax: QUIT

The QUIT statement is used to exit from a statement group defined by a control structure. The QUIT statement is normally used as the object of an IF-THEN statement to specify a conditional exit from a statement group. The statement group defined by the immediately enclosing structure is that from which QUIT exits.

In the case of repetition or loop structures, QUIT passes control to the statement in the line following the delimiter statement that ends the structure. A QUIT from a statement group of an IF structure passes control to the statement in the line after the ENDIF, regardless of whether the statement group followed an IF, ELSEIF or ELSE statement.

The use of QUIT within a GUESS structure is presented with that structure's description.

## 6.4 GUESS-ADMIT-ENDGUESS

Syntax:
```
GUESS
 .
 . statement(s)
 .
ADMIT
 . statement(s)
ADMIT
 . statement(s)
ENDGUESS
```

The GUESS structure provides another form of choice or selection control structure. For much the same reasons that LOOP-ENDLOOP with QUIT is more suitable than WHILE or UNTIL loops in some circumstances, the GUESS structure provides a selection mechanism that is somewhat more flexible than the IF structure. The IF structure requires that control information for selecting a statement group be known on entry to the structure. In many practical situations this ideal availability of information is not attainable.

The GUESS structure permits execution of a statement group to be started on the assumption that it will be the correct choice. In the course of executing a statement group, if this assumption is determined to be false, QUIT may be used to begin executing the next statement group defined by an ADMIT statement. If no ADMIT

statement group follows the current one, QUIT passes control to the statement in the line following the ENDGUESS.

If the end of the statement group is reached (that is, an ADMIT or ENDGUESS statement is encountered), control passes to the statement in the line following the ENDGUESS.

```
GUESS
    IF error THEN QUIT
    .
    .
    IF error THEN QUIT
    .
ADMIT
    .
    . "handle error"
    .
ENDGUESS
```

Situations abound in which an object of data must be analyzed and validated while it is being processed. The above illustrates a technique that can be used to handle errors that are detected at various stages in the processing of a data object.

# Chapter 7

# Procedures and CALL

A program can be organized into components using procedures. A procedure is a group of statements delimited by PROC and ENDPROC statements. Each procedure is identified by a unique procedure name. This name must start with an alphabetic character and can have up to 31 alphabetic characters, digits and underscore ('_') characters. A procedure name is defined with the PROC statement.

A procedure is invoked by referencing its name with a CALL statement. This causes the statements of the procedure to be executed. When the procedure is completed (that is, when the ENDPROC statement is encountered) control returns to the end of the invoking CALL statement. The simple form of the CALL statement is shown below.

```
CALL  procedure-name
```

The corresponding form of procedure definition follows:

```
PROC  procedure-name
   .
   .  statement(s)
   .
ENDPROC
```

Procedure definitions cannot be nested within other procedure definitions or function definitions.

Statements within a procedure can access all variables, matrices, functions, procedures and files defined elsewhere in the program. Procedures can be defined to have formal parameters, that is, variables which have values unique to a particular invocation of a procedure. Initial values of these parameter variables are specified by a parenthesized list of expressions, separated by commas, which follows the procedure name in a CALL statement. This form of the CALL statement is shown below.

```
CALL procedure-name(expression, ..., expression)
```

The corresponding form of procedure definition follows.

```
PROC  procedure-name(variable, ..., variable)
      .
      . statement(s)
      .
ENDPROC
```

When such a CALL statement is executed, parameter-value expressions are evaluated in left-to-right order and formal parameter variables are assigned the positionally corresponding values specified by the expression list. Note that data types of expression values and formal parameter variables must be compatible. Rules concerning this value assignment match those for the LET statement.

The values of variables used as formal parameters for an active procedure apply throughout the program until the procedure is terminated by execution of an ENDPROC statement. When the ENDPROC is executed, variables used as formal parameters are restored to have the values which were in effect prior to executing the CALL statement; control is then returned to the statement following the CALL statement. These rules concerning access to variables and formal parameters are identical to those which traditionally apply to functions in the BASIC language.

One procedure can be CALLed from within another procedure. A procedure can be called from another procedure that was invoked by the first. In fact, a procedure can invoke itself directly. Direct or indirect invocation of a procedure by itself is called *recursion* or *recursive calling*. Note that each recursive activation of a particular procedure will have its own unique formal parameter values.

# Chapter 8

# Functions

Like procedures, functions provide a means of organizing a program into components that can be executed or invoked from various places within the program. A function is invoked by a reference to its name in an expression. Execution of a function results in a value, associated with the function name, which is used in the invoking expression evaluation.

A function name starts with the characters FN, which can be followed by up to 31 alphabetic characters, digits and underscore characters. The FN prefix can be entered in upper case, lower case or mixed case characters. When a program is listed, the prefix characters are always displayed in lower case.

Like variables, a function can be one of three data types, namely, numeric, integer or string. Integer function names end with a percent symbol (%). String function names end with a dollar symbol ($). Numeric function names have no postfix.

A function can be invoked from any valid expression in a program, simply by referencing its name. If parameter values are required by a function definition, these are specified as a parenthesized list of expressions, separated by commas, following the function name. Parameter values and variables are treated in the same manner as those for procedures described in the previous chapter. Similar to procedures again,

functions can be invoked recursively. Likewise, statements within a function definition can access all variables (which are not formal parameters), matrices, functions, procedures and files defined elsewhere in the program.

Two types of function definitions can be used, namely single-line or multi-line functions. The single-line form of function definition follows.

```
DEF  function-name(variable, ..., variable) = expression
```

The parenthesized variable list defines formal parameters and is optional. When such a function is referenced, the expression following the equals symbol is evaluated and returned as the function value to be used in the invoking expression.

The multi-line form of function definition follows.

```
DEF  function-name(variable, ..., variable)
     .
     . statement(s)
     .
FNEND
```

The parenthesized variable list defines formal parameters and is optional. When such a function is referenced, the statements within the function definition are executed. Normally, one of these statements is a LET statement of the following form.

```
function-name = expression
```

The last such LET statement executed defines the value to be returned for use in the invoking expression when the FNEND statement is encountered. If no value is assigned to the function-name a default value is returned. The default value is 0 for numeric and integer functions; it is the null string for string functions.

Note that it is only valid to assign a value to a function name if it corresponds to the currently active function.

# Chapter 9

# Primitive Control

The statements described in this chapter can be used to *control* the order in which the statements of a program are executed. These statements are historically a part of the BASIC language, but are more primitive than the control facilities described in the preceding three chapters. GOSUB and RETURN provide a control facility which is equivalent to CALL and procedures (without parameters). Use of procedures, however, forces a programmer to define clearly the limits of a program component. In addition, one is able to identify a procedure with a name that is indicative of its purpose, in contrast to GOSUB's use of a line number to identify a program component.

## 9.1 GOTO Statement

| | | |
|---|---|---|
| Syntax: | GOTO | line-number |
| or | GO TO | line-number |

Execution of the GOTO statement transfers control to the statement in the line with the specified line number. No statement can follow a GOTO on the same line.

It is an error to exit from an active procedure or function definition using a GOTO. Exit from an active FOR-NEXT terminates the loop, causing the index

control information to be purged.

## 9.2 GOSUB and RETURN

| | | |
|---|---|---|
| Syntax: | GOSUB | line-number |
| or | GO SUB | line-number |
| | RETURN | |

Execution of the GOSUB statement transfers control to the statement in the line specified by the line number. Subsequent execution of a RETURN statement passes control back to the statement following the most recent GOSUB statement executed.

If a FOR-NEXT loop, procedure or function is activated and not completed between execution of the GOSUB and a corresponding RETURN, an error will result.

## 9.3 ON-GOTO, ON-GOSUB

Syntax:

| | |
|---|---|
| ON expression GOTO | line-number, ..., line-number |
| ON expression GO TO | line-number, ..., line-number |
| ON expression GOSUB | line-number, ..., line-number |
| ON expression GO SUB | line-number, ..., line-number |

When one of these statements is executed, the numeric expression is evaluated and rounded to the nearest integer value. This value is used as an index to select a line number from the line number list. Control is then passed to the statement specified by the selected line number, according to the rules of either the GOTO or GOSUB statements, whichever is appropriate. If the integer value resulting from the expression is zero, negative or greater than the number of line numbers specified in the list, an error results. Execution of a RETURN statement subsequent to an ON-GOSUB passes control to the statement following the ON-GOSUB statement.

### 9.4 IF-THEN line number

Syntax: IF expression THEN line-number

This form of the IF-THEN statement is equivalent to the following:

IF expression THEN GOTO line-number

Refer to the chapter concerning Structured Control for an understanding of this statement's meaning.

# Chapter 10

# Input/Output Statements

All useful programs are concerned with the processing of information or data in one form or another. In some cases, a program's data is generated or defined internally, but more frequently data is obtained or *input* from an external source. Input statements can be used to obtain data from the computer's keyboard, from an external storage media, such as disk, or from another computer. Using *output* statements, the results of processing this data can be transmitted to the computer's display screen, to printer paper, to an external storage medium or to another computer.

Both the source of input data and the destination of output data are regarded as *files*. The types of files that can be accessed and their characteristics are described in the Files appendix of this manual. A file to be accessed is identified by its *filename* in an OPEN statement. A filename is specified as a string-valued expression. Valid filename strings are described in the Files appendix of this manual.

Execution of an OPEN statement associates the filename with a *file number*. Other input/output statements identify the file to be accessed using this file number. File numbers are specified as numeric-valued expressions which, when evaluated, are rounded to the nearest integer. File numbers 0 and 1 are reserved for terminal input (keyboard) and output (screen), respectively, and are automatically OPENed by the microBASIC system. Other valid file numbers are the integers in the range from 2 to

32767, inclusive.

The CLOSE statement is used to terminate access to a file and release the file number (so that it can be used with other files). INPUT, LINPUT and GET statements each provide different input facilities. All output is performed with the PRINT statement. Other statements which support the management of files are SCRATCH, RENAME and MOUNT.

## 10.1 OPEN Statement

Syntax: OPEN #file-number, filename, mode

where mode is one of

INPUT, OUTPUT, INOUT or APPEND.

The OPEN statement identifies a file, and performs appropriate initial preparation *for the type of access specified. The file is identified by a filename string expression* and is associated with a positive integer file number. Other input/output statements such as PRINT or INPUT can access a file by specifying this file number.

Some devices that are treated as files do not support both input and output of data. For example, a keyboard supports only input and a printer supports only output. Files stored on disk can be accessed for input or output. The type or mode of access to a file is specified in the OPEN statement.

A mode of OUTPUT or APPEND indicates that data can only be transmitted to the file using the PRINT statement. APPEND mode applies only to files on storage devices such as disk that retain data for subsequent input. OPENing a file in APPEND mode locates the end of existing data in such a file, so that new data is added to its end. OPENing a storage device file for OUTPUT creates a new file, replacing any existing file with the same name. Some storage device files can be OPENed in INOUT mode, allowing both input and output of data. The Files appendix of this manual describes which modes are possible with different types of files.

Examples:

```
OPEN #2, 'Names', INPUT

OPEN #File_count + 2, File_name$, APPEND
```

File numbers 0 and 1 cannot be specified in an OPEN statement. These numbers are reserved for primary terminal input and output, respectively, and are automatically OPENed by the microBASIC system.

## 10.2 CLOSE Statement

Syntax: CLOSE #file-number

The CLOSE statement terminates access to the file associated with the specified file number. The file number is released and can then be associated with other files in subsequent OPEN statements. File numbers 0 and 1 cannot be CLOSEd.

## 10.3 PRINT Statement

Syntax: PRINT #file-number, REC=expression, output-list

where output-list is a series of items separated by commas or semi-colons,

and an item is: expression
MAT matrix-name
or TAB (expression)

The PRINT statement transmits the data values specified in the output list to a file. The output file can be identified by a file number. A file number of 1 is assumed if no file number is specified and the data is displayed on the terminal screen.

For example, the statement

```
print 'Hello'
```

will display the string 'Hello' on a line of the computer's terminal screen. The statement

```
print #2, 'Hello'
```

will transmit the string 'Hello' to the file associated with file number 2 by a prior OPEN statement.

A PRINT statement normally transmits one line or *record* of data to a file. Records are normally PRINTed sequentially, that is, they are transmitted in the order in which they are to appear in the file. The REC= clause can be included in a PRINT

statement to specify the position of the record within the file. Record positioning is only valid for some types of files. The Files appendix of this manual explains which types of files can be accessed in random order. The expression following REC= must have a non-negative numeric value and is rounded to the nearest integer. The value represents a relative record number, the first record in a file being record 0.

```
print #2, rec = 15, 'fifteen'
print #2, 'sixteen'
print #2, rec = 4, 'four'
```

The above three PRINT statements write string values to records 15, 16 and 4, respectively, of the file associated with file number 2. Note that a PRINT statement without a REC= clause, transmits a record sequentially at the position in the file where the last input/output operation finished. Information concerning which types of files can be accessed in this way is contained in the appendix entitled "Files".

Each line or record PRINTed can be viewed as being composed of columns or print zones which are 16 character positions wide. For example, an 80-character line on the terminal screen has 5 such print zones. Each time a comma is encountered in an output-list, space characters are transmitted until the start of the next print zone is reached. This makes it easy to PRINT tables of information with columns that line up vertically.

The statements

```
print 'abc', 'def'
print 'xy', 'z'
```

will display the following two lines on the terminal screen (the first character of each appearing in column 1):

```
abc             def
yx              z
```

A print zone will be skipped if two commas appear in an output list with no item between.

When a semi-colon is used to separate two items in an output list, the second item immediately follows the first without spacing to a new print-zone. For example, the statement

```
print 'abc'; 'def'
```

will display the following line on the terminal screen (the character 'a' appearing in column 1):

```
abcdef
```

If a PRINT output-list ends with a comma or semi-colon, subsequent PRINTs to the same file will continue on the same line or record. A PRINT statement with no output-list can be used to complete a record written using a series of PRINTs ending with a semi-colon or comma. Without preceding uncompleted PRINTs, a PRINT statement with no output-list will write a blank record to the file designated.

```
print 'Hello';
print ' Charlie'
```

The above example displays a line containing the string 'Hello Charlie' starting in column 1.

A TAB item specifies a position within the line or record as a numeric valued expression enclosed in parentheses. This value is rounded to the nearest integer and sufficient space characters are transmitted to reach the specified position. If the value specified is less than one, TAB moves to the first character position of the next line. If the TAB value exceeds the length of the line (record), TAB moves to a position on the following line which is the modulus of the specified value relative to the line size. For example, a TAB value of 165 specified in a PRINT to an 80-column screen positions to column 5 of the next line.

If the TAB value specifies a position within the line that has already been passed by data transmitted, TAB moves to the specified position in the following line. Note that a semi-colon is normally used to delimit a TAB item from the next output-list item, since a comma would position to the start of the print zone following the TAB position.

Numeric values are PRINTed in one of two formats depending upon the magnitude of the number's absolute value. Absolute values which equal or exceed 0.00001 and are less than 1000000000 are printed in the following format

```
   sdb
or sd.fb
```

where "s" is the sign ("-" for negative, blank for positive), "d" is at least one digit representing the integral part, "." is a decimal point, "f" is at least one digit representing the fractional part, and "b" is a blank character. A total of at most 9 significant digits (integral or fractional) are PRINTed. Trailing fractional zero digits

are trimmed as are leading integral zero digits (except one if all integral digits are zero). Examples of values in this format follow.

```
-123456789
  5.83
  0.0000123456789
```

Values which do not fall in the range handled by the first format are printed in normalized scientific format. In this format each number is represented by a sign ("-" or blank), one non-zero integral digit, a decimal point, 8 fractional digits, the letter "E" representing exponent, and a signed two-digit power of 10 scaling factor, followed by a blank character. Examples of this format follow.

```
 1.23456789E-08
-8.42700000E+15
```

If insufficient space remains on a line (record) for a numeric value, transmission of the value begins at the start of the following line. If insufficient space remains on a line for a string value, the characters that will fit are transmitted on one line and the remainder are transmitted at the beginning of the next line.

When a matrix name appears in an output list following the MAT keyword, all elements of the matrix are PRINTed. The elements of the matrix are PRINTed a row at a time, each row starting on a new line (record). A blank line is PRINTed each time a dimension beyond the first two (row and column) is completed. Matrix elements within each row are PRINTed in print zones as if commas were between each. A new line is started for the next item following a matrix in the output list.

If a semi-colon follows the matrix name each element is PRINTed as though it was followed by a semi-colon in an output list.

To understand properly how matrices are PRINTed one should experiment with output to the terminal screen. The characteristics of matrix output to files match those for output to the terminal screen.

## 10.4 MAT PRINT Statement

Syntax:

```
MAT PRINT  #file-number, REC=expression, matrix-list
```

where matrix-list is a sequence of matrix names separated by

commas or semi-colons.

The MAT PRINT statement provides another means of PRINTing all elements of the matrices specified. The characteristics of *MAT PRINT are identical to those of* PRINT for MAT items, except that all items in the list must be matrices. The two statements

```
MAT PRINT A, B
MAT PRINT A; B;
```

are equivalent to

```
PRINT MAT A, MAT B
PRINT MAT A; MAT B;
```

## 10.5 INPUT Statement

Syntax:

INPUT #file-number, REC=expression, prompt-string, input-list

where input-list is a series of items separated by commas, and

an item is:
- variable
- matrix element
- variable substring
- matrix-element substring
- MAT matrix-name
- function name

and prompt-string is a string constant.

The INPUT statement obtains data values from a file and assigns them to the items specified in the input list. The input file can be specified by a file number. A file number of 0 is assumed if no file number is specified and the data is obtained from lines entered using the terminal keyboard. INPUT from the terminal is prompted for by a question symbol, "?", which is displayed on the screen. If a prompt string is specified in the INPUT statement, its characters will be displayed instead of "?".

Data values corresponding to the input list items are scanned from the lines or records of the input file. Each data value is separated from the next by a comma or the end of line (record). As many lines or records will be input and scanned as is

necessary to obtain a value for each input list item. It is an error if data values remain on the current input line following the last data value assigned to an input list item.

INPUT flushes to the end of a record if all data items are not scanned, so that a subsequent INPUT from the same file begins at the start of the next record. A partial record can be INPUT by ending the input-list with a comma. This causes a subsequent INPUT from the same file to resume scanning the same record.

Numeric data values can be in any form that can legally be used to specify a numeric constant in a program. String data can be enclosed in quotation symbol pairs. If string data is not enclosed in quotations, leading and trailing blanks are trimmed, and commas are treated as data value delimiters; that is, if string data is to contain commas, it must be enclosed in quotations. Each value is assigned to its corresponding input list item following rules similar to the LET statement.

An INPUT statement normally accesses the lines or records of a file in the order in which they appear, that is, in sequential order. The REC= clause can be included in an INPUT statement to specify which record should be accessed. Record positioning is only valid for some types of files. The Files appendix of this manual explains which types of files can be accessed in random order. The expression following the REC= must have a non-negative numeric value and is rounded to the nearest integer. The value represents a relative record number, the first record in a file being record 0.

If an INPUT statement attempts to access a record beyond the last record of a file, an end-of-file (EOF) error condition results (see the chapter concerning Error Handling).

When a matrix name appears in an input list following the MAT keyword, it is treated as if each subscripted element of the matrix appeared in the input list row after row.

*Note*:

INPUT from the terminal, without a file-number reference, is treated in the following special way. In conformance with ANS requirements, the number of data values on a line must exactly match the number of input-list items. Errors detected assigning data values to input-list items, or matching the number of values with input-list items, are automatically recovered without interrupting program execution. In such cases, input-list items have their previous values restored (except in the case of entire matrices using the MAT keyword), and INPUT is requested again from the start of the input-list.

## 10.6 MAT INPUT Statement

Syntax: MAT INPUT #file-number, REC=expression, prompt-string, matrix-list

where matrix-list is a sequence of matrix names separated by commas.

The MAT INPUT statement provides another means of assigning values from a file record to all elements of the matrices specified. The characteristics of MAT INPUT are identical to those of INPUT for MAT items, except that all items in the list must be matrices. The statement

```
MAT INPUT A, B
```

is equivalent to

```
INPUT MAT A, MAT B
```

## 10.7 LINPUT Statement

Syntax:

LINPUT #file-number, REC=expression, prompt-string, item

where an item is:
- string variable
- string matrix element
- variable substring
- matrix-element substring
- string function name

The LINPUT statement transmits all characters in a line or record of a file to a string and assigns it to the item specified. Characteristics of the LINPUT statement regarding prompt strings, file numbers and the REC= clause are identical to those of the INPUT statement. Assignment of the character string to the receiving item follows rules similar to the LET statement.

## 10.8 GET Statement

Syntax: GET #file-number, item

where item is:
- variable
- matrix element
- variable substring
- matrix-element substring
- function name

The GET statement is used to obtain characters one at a time from an input file. If the item receiving the character is numeric or integer, the character's numeric code is assigned. This corresponds to the value of the ORD function for the character. GET does not deal with lines or records, simply individual characters.

If GET is accessing the terminal keyboard, the character returned reflects the status of the keyboard at that particular instant. If no key is pressed at that time, a null string or zero numeric value is assigned; otherwise, the character corresponding to the key pressed is assigned. Assignment of the character string or numeric code to the receiving item follows rules similar to the LET statement.

If GET is accessing a file other than the keyboard, there will always be a character to assign, unless the end of the file has been reached.

## 10.9 SCRATCH Statement

Syntax: SCRATCH filename

where filename is a string valued expression

The SCRATCH statement can be used to erase a file from a storage device such as disk. The following examples illustrate the SCRATCH statement.

```
scratch 'names'
scratch OLD_FILE$
```

Refer to the chapter concerning File Management for further information.

## 10.10 RENAME Statement

Syntax: RENAME filename TO name

where filename and name are string valued expressions

The RENAME statement can be used to change the name of a file residing on a storage device such as disk. The following examples illustrate the RENAME statement.

```
rename 'currfile' to 'oldfile'
rename Filename$ to 'temp'
```

Refer to the chapter concerning File Management for further information.

## 10.11 MOUNT Statement

Syntax: MOUNT filename

where filename is a string valued expression

The MOUNT command must be executed with some types of disk units when a new disk is inserted in a disk drive. Refer to the chapter concerning File Management for further information.

## 10.12 RESET Statement

Syntax: RESET

The RESET statement closes all files OPENed by the program.

# Chapter 11

# READ, DATA, RESTORE

DATA statements within a program define what can be thought of as an "internal file". Each DATA statement contains a list of numeric and/or string values which can be accessed and assigned to variables, matrix elements or substrings using READ statements. READ operates with DATA statements in a manner similar to that of INPUT with files. The RESTORE statement can be used to start again at the beginning of the DATA statements.

## 11.1 DATA Statements

Syntax: DATA value-list

where value-list is a sequence of constant values separated by commas

The collection of all DATA statements in a program together form a single list of data values. Each value in this list is separated from the next by a comma or the end of a DATA statement. The collection of DATA statements is ordered by ascending line number, however, other statements can be interspersed between DATA statements.

The constant values in the list can be any valid numeric constant, any valid string constant, or simply a sequence of characters.

Each DATA statement must be on a line by itself; that is, no other statements can precede or follow a DATA statement on the same line. No comments can be placed on the end of a line containing a DATA statement.

Some examples of DATA statements follow.

```
  80 data  5, 18, -3, 'grape'
 900 data  .000483, 9.219e-8, purple
4520 data  This character string is all one string value.
4800 data  "Smith, J."
```

## 11.2 READ Statement

Syntax: READ input-list

where input-list is a series of items separated by commas,

and an item is:
- variable
- matrix element
- variable substring
- matrix-element substring
- MAT matrix name
- function name

The READ statement scans values from the list formed by the collection of DATA statements and assigns them to items specified in the input list. Each READ statement starts scanning the DATA value list where the previous READ finished, whether in the middle of a DATA statement or not. If the DATA statement list does not contain enough values to satisfy a READ input list, an end-of-file (EOF) error condition results (see the chapter concerning Error Handling).

When a matrix name appears in an input list following the MAT keyword, it is treated as if each subscripted element of the matrix appeared in the input list, row after row.

Assignment of each value to its corresponding input list item follows rules similar to the LET statement.

## 11.3 MAT READ Statement

Syntax: MAT READ matrix-list

where matrix-list is a sequence of matrix names separated by commas

The MAT READ provides another means of assigning values from the DATA statement list to all elements of the matrices specified. The characteristics of MAT READ are identical to those of READ for MAT items, except that all items in the list must be matrices. The statement

```
MAT READ A, B
```

is equivalent to

```
READ MAT A, MAT B
```

## 11.4 RESTORE Statement

Syntax: RESTORE

The RESTORE statement resets the READ scanner to the start of the DATA statement value list; that is, the beginning of the first DATA statement the program. This allows the same list of values to be READ more than once. The RESTORE statement has no effect if no DATA statements exist in a program.

# Chapter 12

# Error Handling

Several types of errors can arise during the execution of a program. When an error is detected, a diagnostic message is displayed followed by the text of the line that was being executed. In the case of numeric underflow, numeric overflow and zero division errors, only a message is displayed and the computation continues using a substituted value.

The treatment of errors described above is the normal or default action taken by the microBASIC system in response to errors. The ON-error statements provide a means of having the program handle errors when they occur so that the program does not terminate.

Error conditions are classified in a number of categories. Descriptions of the different classes of errors handled by the ON-error statement follow.

*EOF*: This condition occurs when an input operation attempts to access data past the last record of a file. It also occurs when a READ statement attempts to access values past the end of the DATA statement list. The EOF condition can occur during an output operation if the type of file used cannot be extended and output is attempted beyond the last record of the file.

*IOERR*: This condition occurs when an error arises involving an input/output operation. These errors do not include types of errors covered by other conditions, such as EOF, CONV, etc., which can occur during the execution of an input/output statement.

*ATTN*: This condition occurs when the STOP key is pressed. This is detected after execution of the current *line* is completed and before the next *line* is started.

*SOFLOW*: This condition, string overflow, occurs when the result of a string operation is longer than the maximum string length allowed. In practice, a personal computer's memory space normally will be exhausted before a string this large can be generated.

*OFLOW*: This condition, numeric overflow, occurs when a numeric value is computed that exceeds the magnitude of the largest possible value that can be represented. If the computation is purely integer, the maximum value is 32767 or -32768; otherwise, the maximum is machine infinity. The value of machine infinity can be displayed by printing the value of the INF intrinsic function.

*UFLOW*: This condition, numeric underflow, occurs when a numeric value is computed that has a magnitude smaller than the smallest value that can be represented, machine epsilon. The value of machine epsilon can be displayed by printing the value of the EPS intrinsic function. Numeric underflow cannot occur with purely integer computations.

*ZDIV*: This condition, zero divide, occurs when a numeric value is divided by zero.

*CONV*: This condition, conversion error, occurs when it is attempted to convert a sequence of characters to a numeric value and the characters do not form a valid numeric constant specification. This error can occur when processing values for numeric items in INPUT and READ statement input lists. This condition can also occur when converting a string value to a numeric value with the VALUE intrinsic function.

*ERR*: This condition, general error, occurs when any error occurs that does not fit one of the previously described error classes. For example, an error of this class occurs when a matrix subscript value is used which exceeds the defined dimension of the matrix.

## 12.1 ON-Error IGNORE

Syntax: ON error-condition IGNORE

Some classes of errors can be ignored by executing an ON-Error IGNORE statement. Error-conditions that *can* be ignored are EOF, IOERR, OFLOW, UFLOW, ZDIV and CONV. In particular, it is common to ignore EOF and IOERR, and to query the value of the IO_STATUS intrinsic function after each input/output operation. The following example illustrates a program that loops reading names from a file and displaying them on the terminal screen.

```
10  open #2, 'names', input
20  on eof ignore
30  loop
40      linput #2, name$
50      if io_status <> 0 then quit
60      print name$
70  endloop
80  close #2
```

A name is input from each record of the file called 'names'. When all names have been input and displayed, the next LINPUT statement will fail with an EOF error. Since the ON EOF IGNORE statement has been executed, the program continues. The IO_STATUS intrinsic function reflects the status of the latest input/output operation. The value of io_status is 0 after each successful input of a name record. Its value becomes 2 when an EOF failure occurs. Refer to the appendix of this manual concerning intrinsic functions for a fuller description of the IO_STATUS function. In the example program, an exit is taken from the loop when EOF is detected in line 50.

## 12.2 ON-Error SYSTEM

Syntax: ON error-condition SYSTEM

When this statement is executed, the handling of the class of error specified reverts to the default action normally taken by the microBASIC system.

## 12.3 ON-ENDON Structure

Syntax: ON error-condition
.
. statement(s)
.
ENDON

Execution of this type of ON-error statement defines a group of statements that are to be executed when the specified error occurs. If such an ON structure is defined when its type of error occurs, control is transferred to the statement on the line following the ON statement. After appropriate error recovery, the program must be terminated with a STOP statement or control passed back with the RESUME statement. It is an error to execute the ENDON statement.

## 12.4 RESUME Statement

Syntax: RESUME
or RESUME NEXT

The RESUME statement can only be executed from within an ON-ENDON structure that was invoked because of an error condition. The RESUME statement transfers control to the beginning of the *line* containing the statement that was executing when the error was detected. The RESUME NEXT statement transfers control to the *line* following that in which the error was detected. Note that use of multiple statements per line should be avoided or carefully considered when using these error handling facilities.

# Chapter 13

# CHAIN and USE

The CHAIN statement can be used to load another program, replacing the current one, and transfer control to its first statement. Employing this facility, systems can be developed consisting of several programs which collectively would be too large to fit in the computer's workspace.

A list of data values can be passed, like parameters, from the CHAINing program to the new program that is loaded from a file. These values are assigned to variables specified in a list when the USE statement is executed within the new program.

There is no means of returning from the new program to the end of the CHAIN statement that invoked it, since the CHAINing program is no longer in the workspace. The only means of re-invoking the first program is to CHAIN back to it.

## 13.1 CHAIN Statement

Syntax: CHAIN filename,FILES,expression-list

where filename is a string-valued expression designating the name of a file containing a BASIC program.

and expression-list is a list of expressions separated
by commas or a single keyword, NAMES.

When a CHAIN statement is executed, its filename expression is evaluated, the list of expression values are computed and saved, the workspace is cleared and a program is loaded from the file identified by the filename. CHAIN attempts to load a program file with the specified name as created with the STORE command. If this fails, CHAIN attempts to obtain the program source from a data file with the specified name, as created with the SAVE command. All variables and matrices of the new program are initialized as if the program was started with the RUN command. If NAMES is specified instead of a list of expressions, all variables and matrices of the program are passed to the new program and their values are assigned to variables and matrices with the same names.

If the optional FILES keyword is specified, all active files remain open and can be referenced by the new program using the same file numbers. Otherwise, any outstanding open files are closed before the new program is loaded.

Three examples of the CHAIN statement follow.

```
CHAIN 'Menu'
CHAIN 'Overlay', FILES, NAMES
CHAIN 'Update', Customer_name$, Customer_number, Payment
```

## 13.2 USE Statement

Syntax: USE variable, ..., variable

Execution of the USE statement assigns the values from the CHAIN list to positionally corresponding variables in the USE list. The USE statement appears in the program invoked by the CHAIN statement. The number of values passed must match the number of items in the USE list. Data types of the CHAIN values and corresponding USE variables must be compatible according to the assignment rules of the LET statement. An example follows which corresponds to the last example of CHAIN in the previous section.

```
USE Name$, Number, Amount
```

# Chapter 14

# Miscellaneous Statements

This chapter describes statements not covered in other chapters.

## 14.1 STOP Statement

Syntax: STOP

The STOP statement terminates execution of the program. Control information concerning return locations for any outstanding CALLs, GOSUBs or function calls is purged. Execution of a program cannot be resumed with the CONTINUE command following execution of a STOP statement. Any outstanding open files are closed.

## 14.2 PAUSE Statement

Syntax: PAUSE

The PAUSE statement interrupts execution of a program. Control information concerning outstanding CALLs, GOSUBs and function calls is preserved. The status and position of open files is also preserved. When the PAUSE statement is executed, the line containing it is displayed on the terminal screen and execution is suspended.

microBASIC commands can be entered while the program is suspended, or immediate-mode statements can be executed. The program can be edited (changed) while suspended. Refer to the chapter entitled Debugging Programs for information concerning possible side effects of editing. The CONTINUE command causes execution of the program to resume at the start of the *line* following the PAUSE statement. The GOTO statement can be used in immediate-mode to resume execution at some other line.

## 14.3 RANDOMIZE Statement

Syntax: RANDOMIZE

The RANDOMIZE statement is used to generate an unpredictable seed for random number generation using the RND intrinsic function. The RND function will generate the same sequence of values each time a program is run if the RANDOMIZE statement is not used.

## 14.4 POKE Statement

Syntax: POKE address, expression, ..., expression

where address is a numeric valued expression

The POKE statement can be used to store values directly into the computer's memory at the address specified. This address can also specify device registers which are addressable by the computer. The numeric expression specified for the address is evaluated and rounded to the nearest integer between 0 and 65535 inclusive. Negative address values in the range -32768 to -1, inclusive, are considered as 16-bit unsigned binary values and are treated as addresses in the range 32768 to 65535. Although it is possible to specify addresses in this range with negative values, it is usually more convenient and simpler to use positive address values.

The expression list following the address must result in numeric values which, when rounded to integers, fall in the range 0 to 255 inclusive. These values are stored as 8-bit binary numbers in the byte at the address specified and in subsequent bytes.

The POKE statement provides a very primitive and powerful capability. It can be used to manipulate the computer's memory and devices directly to perform clever tricks. However, *caution* should be exercised since random POKEs can clobber important system information or put the microprocessor in a state such that programs can no longer run without restarting the computer. Programmers using POKE should

understand fully the hardware configuration of their personal computer and the system software's use of locations in memory. POKE users must assume responsibility for any strange behaviour by the system software that results.

Note that an intrinsic function, PEEK, allows bytes of memory to be examined. PEEK is described with the other intrinsic functions in an appendix of of this manual.

## 14.5 SYS Statement

Syntax: SYS address

where address is a numeric valued expression

The SYS statement can be used to invoke a system program or other machine-language program at the address specified. Control can be returned to the BASIC program at the statement following the SYS by executing a machine-language RTS (return from subroutine) instruction.

# Appendices

# Appendix A

# Command Language Summary

This appendix summarizes the syntax of commands in Waterloo microBASIC. It is included for easy reference. Terms used, such as filename, and details of each particular command's operation should be obtained by reference to the pertinent section of the manual.

## A.1 Notation

Some commands have optional components. These are depicted by enclosure within square brackets. The following example indicates that a 'filename' is optional in the RUN command.

RUN ['filename']

In addition, several commands may be abbreviated. The optional portion of a command name is also enclosed in square brackets. For example,

L[IST]

indicates that the LIST command can be typed as L, LI, LIS, or LIST.

## A.2 System Command Summary

(1) A[UTOLINE] [line number][,increment]

(2) BYE

(3) CLEAR

(4) CONT[INUE]

(5) DEL[ETE] line range

(6) DI[RECTORY] ['filename']

(7) L[IST] [line range]

(8) LOAD 'filename'

(9) MERGE 'filename'

(10) MOUNT 'filename'

(11) OLD 'filename'

(12) RENAME 'filename' TO 'name'

(13) RENUM[BER] [line number][,increment]

(14) RUN ['filename']

(15) SAVE [line range] 'filename'

(16) SCRATCH 'filename'

(17) SETUP

(18) STEP [line number]

(19) STO[RE] 'filename'

(20) TYPE 'filename'

# Appendix B

# Programming Language Summary

This appendix briefly describes some general characteristics of the Waterloo microBASIC programming language. It is included for easy reference and is not a full description of the language.

## B.1 Line Numbers

- must appear at beginning of each program line
- integral value in range 1 to 65529
- leading zeroes ignored

## B.2 Spacing

- required to separate keywords from names, numeric constants and other keywords
- cannot appear inside names, keywords, numeric constants or multi-character operators

### B.3 Comments

- comments can appear at end of most lines
- comments begin with an exclamation mark, '!'
- everything after the ! is taken as documentation
- the REM statement may also be used to enter comments; the statement begins with the keyword REM and the rest of the line is taken as comments or remarks.

### B.4 Multiple Statements Per Line

- multiple statements can be entered on one line separated by colons, ':'
- this is not recommended since it renders programs less readable
- after error interrupts, RESUME and CONTINUE start at the beginning of the line, even if the error did not occur in the first statement of the line
- structured control statements must appear alone on a line, e.g., IF, ELSE, etc.
- DATA statements must appear alone on a line and cannot be followed by comments

### B.5 Multiple-Line Statements

- statements can span multiple lines
- last non-blank character of the line to be continued must be an ampersand, '&'
- first non-blank character following the line number of the continuing line must be an ampersand
- keywords, names, constants or operators cannot span lines
- DATA statements cannot be continued

## B.6 Names

- sequence of 1 to 31 characters starting with an alphabetic character and consisting of alphabetic characters, digits and the underscore character '_'
- function names have prefix 'fn' or 'FN'
- variables and functions can be of 3 types, namely, string, integer and floating-point
- string variable and function names have a $ postfix, e.g., course$, fn grade$
- integer variable and function names have a % postfix, e.g., age%, fn_enrollment%
- floating-point variable and function names have no postfix, e.g., mark, fn_average
- procedure names have no prefix or postfix
- matrix names have the same postfixes as variables and functions for data types integer, floating-point and string
- an array can have the same name as a variable; they are distinguished one from another by usage

## B.7 Uppercase/Lowercase Alphabetics

- uppercase and lowercase alphabetic characters are treated as equivalent within keywords and intrinsic function names; e.g., IF,If,if, and iF are equivalent and are displayed in lowercase ('if') by the LIST command
- uppercase and lowercase letters are distinguished between in names of variables, arrays, functions and procedures; e.g., Name is separate and distinct from name
- uppercase and lowercase letters are distinguished between in string constants or literals; e.g., "Abc" is different from "ABC"
- uppercase and lowercase letters are distinguished between in filenames for the Commodore disk; case sensitivity in filename specifications for a

specific computer is defined by the rules of the operating system being used.

## B.8 Expression Evaluation

| *Priority* | *Operation* | *Example* |
|---|---|---|
| 11 | matrix subscripting | table(i,j) |
| | function reference | fna(x,y) |
| 10 | substring | name$( start : end ) |
| 9 | enclosed in parentheses | 3 * ( i+1 ) |
| 8 | exponentiation ↑, ** | value ↑ 3 |
| 7 | unary plus + | +5 |
| | unary minus - | -2 |
| 6 | multiply * | quantity * unit_cost |
| | divide / | gallons / 4 |
| 5 | addition + | total + amount |
| | subtraction - | total - credit |
| | concatenation + | province$ + ", Canada" |
| 4 | relational operators =,>,<,<>,>=,<= | total >= 50 |
| 3 | logical NOT | NOT age% > 30 |
| 2 | logical AND | price > 10 AND quantity < 3 |
| 1 | logical OR | errors > 5 OR average < 50 |

## B.9 Intrinsic Functions

| *Function* | *Meaning* |
|---|---|
| abs(x) | returns the absolute value of parameter x |
| atn(x) | returns the arc TANGENT (in radians) of the parameter x, where -PI/2 < atn(x) < PI/2 |
| chr$(i%) | returns a single-character string which represents the character in position i% of the character set defined by the system |
| cos(x) | returns the COSINE of angle x; x is expressed as the number of radians |

| | |
|---|---|
| cursor(i%) | sets cursor on terminal screen to position i% if value in range 1 through 2000 (25 rows by 80 columns). Returns cursor position whether set or not. |
| date$ | returns a string value which is the current date. |
| eps | returns the value of "machine epsilon", that is, the smallest numeric value that can be represented by the computer being used |
| exp(x) | returns the value of Euler's constant e, raised to the power of the parameter x |
| fp(x) | returns the fractional part of the parameter value x; the result has the same sign as x |
| free | causes available memory to be coalesced and returns the number of bytes remaining free |
| hex(s$) | returns a numeric value corresponding to the hexadecimal value represented by the characters of string s$; hexadecimal values are defined in terms of the digits 0-9 and letters A-F |
| hex$(i%) | returns a string value which represents the value of parameter i% in hexadecimal, base 16; hexadecimal values are defined in terms of the digits 0-9 and letters A-F |
| idx(a$,b$) | returns a number representing the position (origin 1) at which the character string b$ first occurs in the character string a$. Zero is returned if b$ is not found in a$. |
| inf | returns the value of "machine infinity", that is, the largest numeric value that can be represented by the computer being used |
| int(x) | returns the largest integer which is not greater than floating-point parameter value x |
| io_status | returns status of last input/output operation; value 0 indicates success; 1 indicates end-of-record (only after a GET statement); 2 indicates end-of-file; 3 indicates an input/output error. |

| | |
|---|---|
| io_status$ | returns a string value corresponding to the status of the last input/output operation: a null string indicates success (including the end-of-record status); "eof" indicates end-of-file; a message text describing the error is returned in other cases (this text is dependent upon what devices are being used and upon which system the program is being executed) |
| ip(x) | returns the integer part of the floating-point number x; the result has the same sign as x |
| len(s$) | returns the length of string parameter s$ |
| log(x) | returns the natural logarithm (base e) of the parameter x; x must have a positive value |
| mod(x,y) | returns the modulus of the number x for the range y. The calculation is equivalent to specifying x-y*INT(x/y). |
| ord(s$) | returns the position, or ordinal value, of character s$ in the set of characters defined by the system; s$ must be a one-character string |
| peek(i%) | returns an integer representing the value stored in the byte located at address i% in the computer system. |
| pi | returns the value of the mathematical constant pi (approximately 3.14159265) |
| rnd(x) | returns a pseudo-random real number in the range (0,1) according to a uniform distribution over this interval. When no parameter is specified, the value is computed from the last value according to a fixed algorithm. When a parameter x is specified, the random number generator is reset using the parameter x as a seed (starting point). A reproducible sequence of random numbers can be generated by initially using a seed and then successively invoking the rnd function without a parameter. An unpredictable starting point can be set with the RANDOMIZE statement. |
| rpt$(s$,n) | returns a string consisting of the string s$ repeated (concatenated) n times. Value n is rounded to an integer if necessary. |

| | |
|---|---|
| sgn(x) | returns a value based on the sign of numeric parameter x: -1 if x<0; 0 if x=0; 1 if x>0 |
| sin(x) | returns the SINE of angle x; x is expressed as the number of radians |
| sqr(x) | returns the square root of the number x; x must be a non-negative value |
| str$(a$,s,l) | returns a string value which is the portion (substring) of the string a$, starting at characters position s with length l characters. Values s and l are rounded to integers if necessary. If l is less than one after rounding, a null string is returned. When fewer than l characters exist from position s onward, the remainder is returned. |
| tan(x) | returns the TANGENT of the angle x ; x is expressed as the number of radians |
| time | returns the current time as number of seconds since midnight. |
| time$ | returns a string value which is current time of day. |
| value(s$) | returns a numeric value corresponding to the number defined by the characters of string parameter s$ |
| value$(x) | returns a string value representing the value of numeric parameter x as it would be displayed by a PRINT statement without leading or trailing blanks |

## B.10 Statement Summary

### B.10.1 Notation

- keywords appear in uppercase letters
- optional phrases or components appears enclosed in square brackets [ ]
- a choice of phrases or components appears in a vertical list enclosed in braces { }
- several short forms are used for terms as follows

| *Short Form* | *Term* |
|---|---|
| matrix-elmt | subscripted element of an matrix |
| matrix-name | name of matrix |
| condition | conditional expression (i.e., numeric expression where zero is false and non-zero is true) |
| file-name | string-valued expression which represents the name of a file |
| file-num | integer-valued expression which designates a file by number |
| float-exp | floating-point valued expression |
| func-name | name of user-defined function |
| int-exp | integer valued expression |
| line-num | line number |
| num-matrix | numeric matrix |
| num-con | numeric constant |
| num-exp | numeric expression (integer or floating-point) |
| num-var | numeric variable name |
| proc-name | procedure name |
| prompt | string-valued constant displayed to prompt for input |
| string-con | string-valued constant |
| string-exp | string-valued expression |
| string-func-name | string-valued function name |
| string-matrix | string matrix |
| string-matrix-elmt | string matrix element |
| string-var | string variable name |
| var-name | variable name |
| var-substr | substring of variable or matrix element |

### B.10.2 Statements

(1) CALL proc-name [(param[,param]...) ]

where param is one of:
{int-exp}
{float-exp}
{string-exp}

(2) CHAIN file-name {, [,FILES][,param][,param]... }
{, [,FILES][,NAMES] }
{, [,NAMES][,FILES] }

where param is one of:
{int-exp}
{float-exp}
{string-exp}

(3) CLOSE #file-num

(4) DATA const[,const][,const]...

where const is one of:
{num-con}
{string-con}

(5) DEF func-name[(var-name[,var-name]...)]=exprn

where exprn is num-exp or string-exp

(6) DEF func-name[(var-name[,var-name]...)]

. [statement(s)]

[LET] func-name = {num-exp}
{string-exp}

FNEND

(7) DIM matrix-name(n1[,n2]...) [,matrix-name(n1[,n2]...)]

where n1,n2... are int-exp

(8) END

(9) FOR num-var = num-exp TO num-exp [STEP num-exp]
.
. [statement(s)]
.
NEXT num-var

(10) GET [#file-num,] {var-name}
{matrix-element}
{var-substring}
{func-name}

(11) GO[ ]SUB line-num

(12) GO[ ]TO line-num

(13) GUESS
.
. [statement(s)]
.
[ADMIT
.
. [statement(s)]
.]
ENDGUESS

(14) IF condition THEN {line-num}
{statement}

(15) IF condition
.
. [statement(s)]
.
[ELSEIF condition
.
. [statement(s)]
.]
[ELSE
.
. [statement(s)]
.]
ENDIF

(16) INPUT [#file-num,] [REC=num-exp,] [prompt,] input-list

where input-list is:

```
var-name [,var-name ]
matrix-elmt [,matrix-elmt]
var-substr [,var-substr] . . .
func-name [,func-name]
MAT matrix-name [,MAT matrix-name]
```

(17) [LET] {var-name} {int-exp}
{matrix-elmt}= {float-exp}
{var-substr} {string-exp}
{func-name}

(18) LINPUT [#file-num,][REC=num-exp,][prompt,] {string-var}
{string-matrix-elmt}
{var-substr}
{string-func-name}

(19) LOOP

. [statement(s)]

ENDLOOP

(20) LOOP

. [statement(s)]

UNTIL condition

(21) MAT num-matrix = (num-exp)
MAT string-matrix = (string-exp)
MAT num-matrix = num-matrix
MAT string-matrix = string-matrix
MAT num-matrix = (num-exp) + num-matrix
MAT num-matrix = (num-exp) * num-matrix
MAT num-matrix = num-matrix + num-matrix
MAT num-matrix = num-matrix - num-matrix
MAT num-matrix = num-matrix * num-matrix
MAT num-matrix = ZER
MAT string-matrix = NULL$
MAT num-matrix = IDN
MAT num-matrix = TRN(num-matrix)

(22) MAT INPUT [#file-num,][REC=num-exp,][prompt,]
matrix-name [,matrix-name]...

(23) MAT PRINT [#file-num,][REC=num-exp,]
matrix-name [,matrix-name]...

(24) MAT READ matrix-name [,matrix-name]...

(25) MOUNT file-name

(26) ON err-condn
.
. [statement(s)]
.
RESUME [NEXT]
.
ENDON

where err-condn is: {EOF}
{IOERR}
{ATTN}
{SOFLOW}
{OFLOW}
{UFLOW}
{ZDIV}
{CONV}
{ERR}

(27) ON err-condn SYSTEM

(28) ON err-condn IGNORE

valid only for EOF,IOERR,OFLOW,UFLOW,ZDIV,CONV

(29) ON num-exp GO[ ]SUB line-num[,line-num]...

(30) ON num-exp GO[ ]TO line-num[,line-num]...

(31) OPEN #file-num, file-name,{ INPUT}
{ OUTPUT }
{ INOUT }
{ APPEND }

(32) OPTION { BASE 0 }
{ BASE 1 }

(33) PAUSE

(34) POKE num-exp,num-exp [,num-exp ]...

where the first num-exp is the address of the first byte to be changed and the other num-exp's are the value(s) to be stored in this byte and bytes following

(35) PRINT [#file-num,][REC=num-exp] output-list

where output-list is: [item] [, item ] [;]...

and item is: num-exp
string-exp
MAT matrix-name
TAB (num-exp)

*Note*:

A question symbol, ?, may be typed in place of the keyword PRINT.

(36) PROC proc-name [(var-name[,var-name]...)]
.
. [statement(s)]
.
ENDPROC

(37) QUIT

can appear within IF-ELSEIF-ELSE-ENDIF
LOOP-ENDLOOP, LOOP-UNTIL
WHILE-ENDLOOP, WHILE-UNTIL
GUESS-ADMIT-ENDGUESS
FOR-NEXT

(38) RANDOMIZE

(39) READ input-list (see INPUT for input-list)

(40) REM [comment]

(41) RENAME file-name TO file-name

(42) RESTORE

(43) RESUME [NEXT] (see ON-ENDON)

(44) RETURN (after GOSUB)

(45) SCRATCH file-name

(46) STOP

(47) SYS num-exp

where num-exp is the address of a system routine
or user-written machine-language routine to be called

(48) USE var-name [,var-name ]
(use CHAIN params)

(49) WHILE condition
.
. [statement(s)]
.
ENDLOOP

(50) WHILE condition
.
. [statement(s)]
.
UNTIL condition

## B.11 Keywords

- keywords are words with special meaning in the programming language
- cannot be used as names of variables, matrices, functions or procedures
- can be entered in lower or upper case or in combination of both; e.g., Print, print, PRINT
- always listed in lower case due to statement encoding
- consist of words which define statement types, (e.g., DATA, PRINT), special words used in statements (e.g., STEP, IGNORE) and intrinsic function names (e.g., SIN, ORD).

abs
admit
and
append
atn
attn
base
call
chain
chr$
close
*conv*
cos
data
date$
def
dim
else
elseif
end
endguess
endif
endloop
endon
endproc
eof
eps
err
exp
files
fnend
for
fp
free
get
go
gosub
goto
guess
hex
hex$
idn
idx
if
ignore
inf
inout
input
int
io_status
*io_status$*
ioerr
ip
len
let
linput
log
loop
mat
mount
names
next
not
null$
oflow
on
open
option
or
ord
output
pause
peek
pi
poke
pos
print
proc
quit
randomize
read
rec
rem
rename
restore
resume
return
rnd
rpt$
*scratch*
sgn
sin
soflow
sqr
step
stop
str$
sub
sys
system
tab
tan
then
time
time$
to
trn
uflow
until
use
using
value
value$
while
zdiv
zer

# Appendix C

# Files

Some devices, such as disk, provide external storage in which files may be kept. Copies of programs may be stored in these files. In addition, files may be created and accessed by programs for the purposes of storing, retrieving and updating data. A file might contain data such as marks for the students in a particular class. Data in files is retrieved with *input* operations and stored or updated with *output* operations. Devices which provide file storage are referred to as *file-oriented devices*.

Each file is identified by a name. Since files may be stored on different devices, it is sometimes necessary to specify a device and name to identify a particular file. Some devices support data input and/or output operations, but do not provide storage for retaining files. For example, data can be output on a printer or display screen and can be input from a keyboard. Input and output operations with such devices are similar to input and output with files. Consequently, such devices are treated as special files. The term *filename*, as used in the context of this manual, includes the specification of a device, a name of a file, or both.

File support, in general, is described in the System Overview manual. The reader should read this manual for a complete description of file types and devices available. Waterloo microBASIC supports both text and fixed file types and allows sequential and relative record access. Variable files are not supported by Waterloo microBASIC.

Waterloo microBASIC supports disk, printer, terminal, keyboard, serial and host devices. Only the GET statement should be used to input characters from the keyboard device. Relative record access can be used to display data on specific lines of the terminal screen.

Using the Commodore disk, only files of type fixed and format REL can be accessed by relative record number. Only files of format REL can be OPENed with INOUT access mode. Only files of format SEQ can be OPENed with APPEND access mode.

Waterloo microBASIC supports input/output with host computer files using device name "host" and the host communications support of the Waterloo System Library. Rules concerning relative or sequential access and file OPEN modes depend upon the file capabilities provided by the host computer system.

Waterloo microBASIC is an interactive BASIC language interpreter which provides simple, comprehensive facilities for entering, running, debugging and editing programs. Waterloo microBASIC includes ANS BASIC as defined in the 1978 X3.60 standard with one minor exception. The programming language supports many important extensions beyond standard BASIC. These include:

- An extensive set of control statements to facilitate Structured Programming
- Long names for variables and other program entities
- Procedures that can be CALLed with parameters
- Multi-line function definitions with numeric, integer and string results
- Sequential and relative (random) input/output
- True integer arithmetic and bit-oriented logical operations using integers
- MAT statements supporting operations on entire matrices
- Powerful character-string manipulation features
- A broad range of intrinsic functions
- Interactive debugging facilities

This manual is divided into four major components:

- An introduction to the general characteristics of the system including a series of annotated examples
- A comprehensive reference guide describing the Command Language
- A comprehensive reference guide describing the Programming Language
- Appendices containing summaries of both the Command and Programming Languages, and describing use of files with Waterloo microBASIC.

DISTRIBUTED BY

Howard W. Sams & Co., Inc.

4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

$10.95/21906

ISBN: 0-672-21906-9