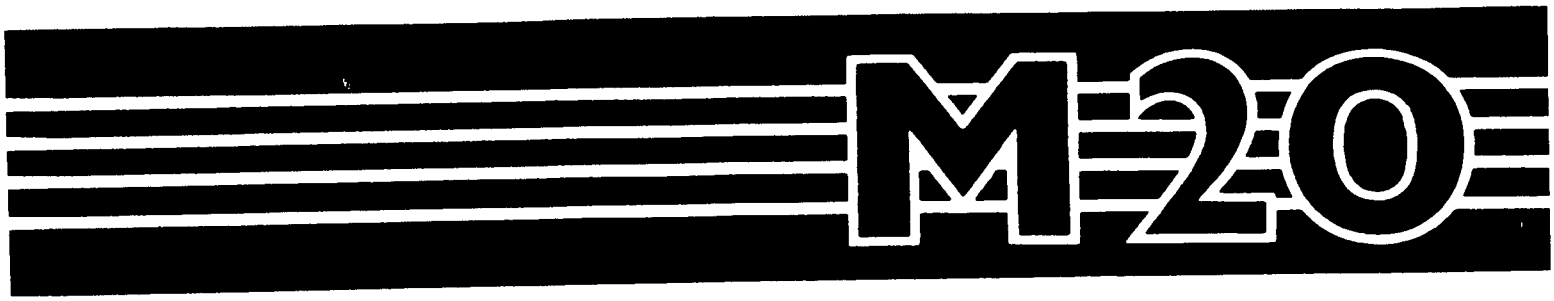
BASIC Language

Reference Manual

olivetti L1



BASIC Language

Reference Manual

olivetti L1

PREFACE

This is a simple guide to the use of BASIC on the OLIVETTI M20 System. It introduces the reader to BASIC, with the help of many figures, tables and examples. Related statements and commands are dealt with in the same chapter.

ALL BASIC STATEMENTS, COMMANDS, AND FUNCTIONS ARE LISTED IN ALPHABETICAL ORDER IN APPENDIX E, FOR SPEEDY REFERENCE.

Previous programming experience is not strictly required. Only a basic knowledge of data processing is assumed.

Related Publications

L1 M20
Professional Computer Operating System (PCOS) User Guide

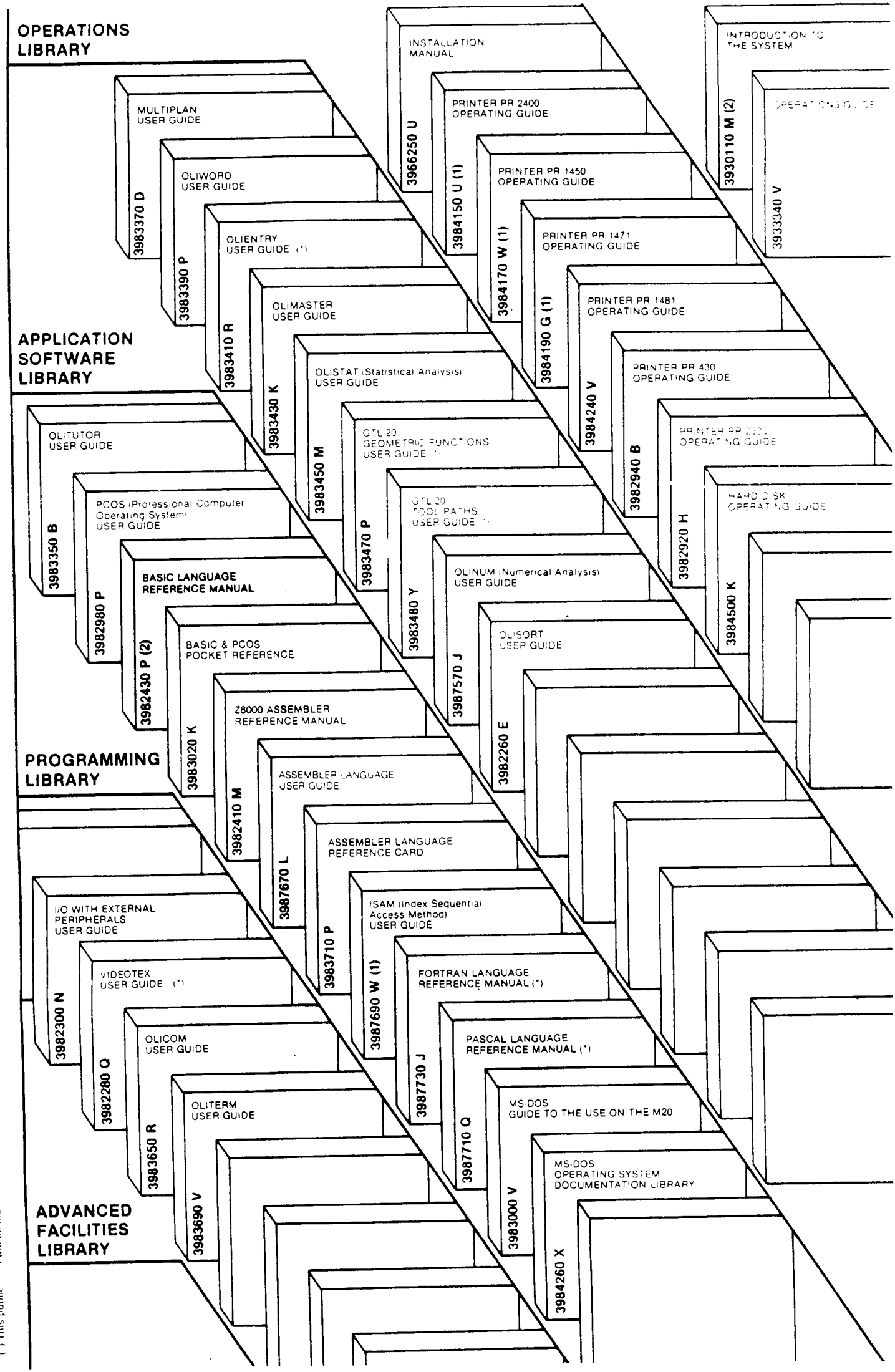
DISTRIBUTION: General (G)

THIRD EDITION: December 1982

RELEASE: 1.3 onwards

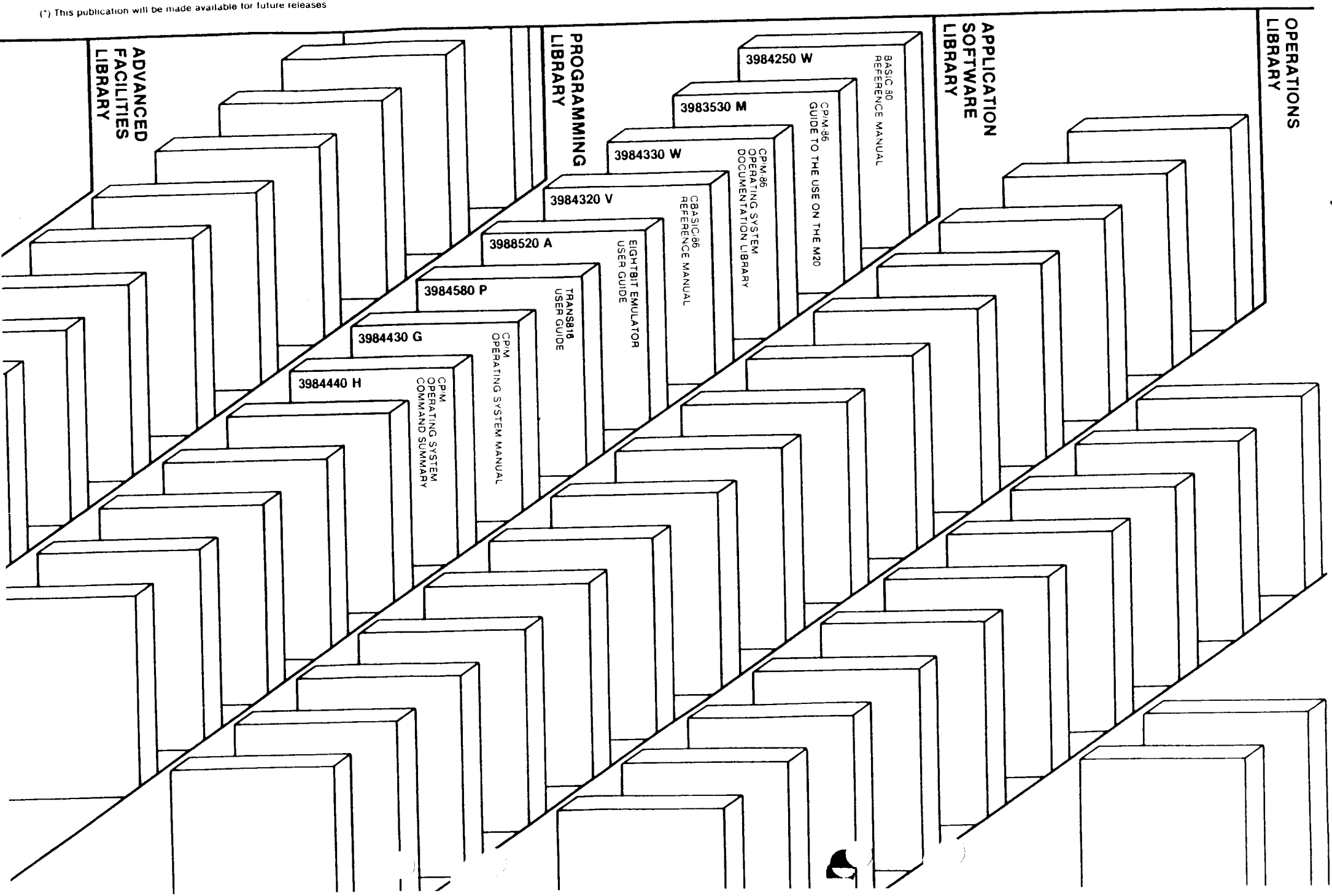
PUBLICATION ISSUED BY:

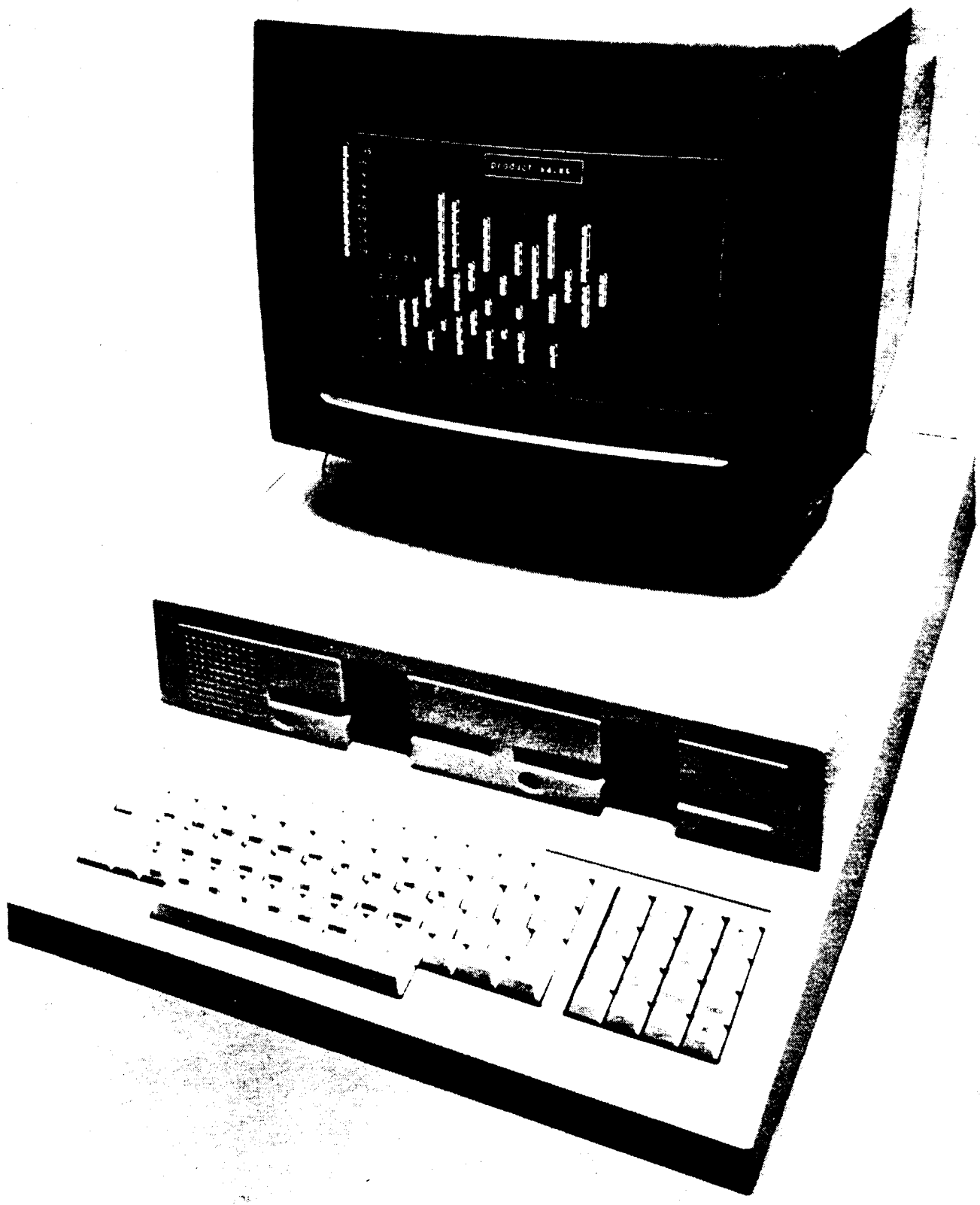
Ing. C. Olivetti & C. S.p.A.
Servizio Centrale Documentazione
77, Via Jervis-10015 IVREA (Italy)



(*) This publication will be made available for future releases

(*) This publication will be made available for future releases





CONTENTS

3. UPDATING AND MODIFYING A PROGRAM

4. DATA

CONTENTS

CONTENTS

CONTENTS

1. WHAT IS BASIC?

ABOUT THIS CHAPTER

This chapter introduces you to the Model 20 (M20) BASIC language. It illustrates the PCOS (Professional Computer Operating System) and BASIC environments, and the use of the Keyboard. Moreover, it tells the user how to use the machine as a calculator, how to enter and run a program, and the modes of operation in BASIC.

CONTENTS

WHAT IS BASIC?

THE BASIC LANGUAGE

BASIC (Beginner's All-purpose Symbolic Instruction Code) is a general purpose high-level programming language.

YOU CAN USE BASIC TO SOLVE BOTH BUSINESS AND SCIENTIFIC PROBLEMS.

BASIC IS EASY TO LEARN AND USE, AS IT CONSISTS OF SELF-EXPLANATORY STATEMENTS AND COMMANDS.

Different BASIC versions are available on different computers. The first was developed at Dartmouth College by John G. Kemeny and Thomas E. Kurts. From now on, when we speak of BASIC we refer to the version used on the Model 20 (M20).

THE M20 BASIC IS A MICROSOFT BASIC VERSION, EXTENDED WITH GRAPHICS AND IEEE 488 STANDARD INTERFACE PACKAGES.

PCOS AND BASIC ENVIRONMENTS

The M20 System may be simply defined as a computer and a set of programs supplied by Olivetti. These "System Programs" are resident on a 5 1/4 in. floppy disk (system disk). Theymay be loaded onto the hard disk in an M20 hard disk system.

The System Programs, which include PCOS and BASIC, permit you to instruct the computer in a manner similar to human language. They work by converting your instructions into a machine-language understood by the computer itself. You interact with the computer using PCOS and BASIC commands and sets of statements referred to as BASIC programs.

Note: From now on we shall use:

- diskette instead of 5 1/4 in. floppy disk, for brevity;
- disk instead of either a diskette or the hard disk.

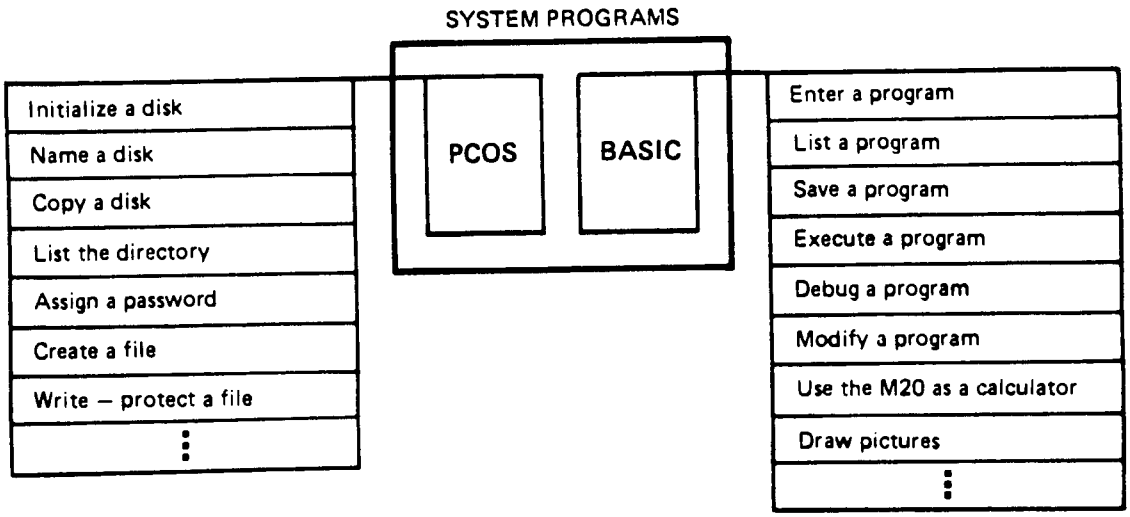


Figure 1-1 System Programs

USING THE KEYBOARD

The keyboard allows entry of all the standard text and control characters.

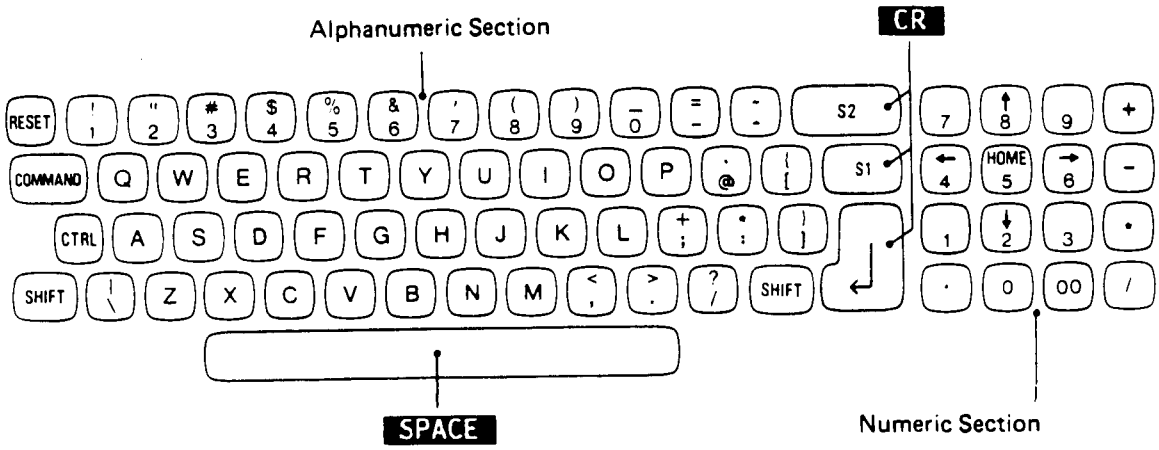


Figure 1-2 The Keyboard (USA-ASCII Version)

Note: All the characters shown in this manual refer to the USA-ASCII Keyboard. Appendix B shows the national equivalents for those ASCII characters which will appear on the display screen or printer.

When we want to specify the keys the user must press to perform a certain action, we shall show the exact sequence of keys in reverse (white on black): the keys illustrated in reverse in figure 1-2 are also included in this sequence; for example:

L **I** **S** **T** **CR**

By convention, we use **CR** to indicate any of the three carriage return/line-feed keys (**S1**, **S2**, and ↵), **SPACE** to indicate the space bar, and **SHIFT** to indicate either of the two shift keys. (Only USA ASCII and USA-ASCII with BASIC verbs keyboards have SHIFT written on the keys).

When we want to remind the user to press **CR** to send a line to the system, we shall show **CR** at the end of the line. For example:

DELETE 1ØØ - 2ØØ **CR**

ENTERING CHARACTERS

IF...	THEN...
you press a key (or a combination of keys)	the character it represents is immediately shown on the screen. When characters are being entered, the blinking cursor (▮) indicates the position the next character will occupy
you want to enter a lower case character or the lower symbol on those keys containing two symbols	just press the key, e.g. **A** for a
you want to enter upper case characters or the upper symbol on these keys containing two symbols	hold down one of the two **SHIFT** keys and press the corresponding key, e.g. **SHIFT** **A** for A. YOU MAY ENABLE OR DISABLE SHIFT LOCK FOR LETTERS (A-Z) BY PRESSING **COMMAND** **?/** (see Control Characters below)
you want to enter a number	use either the top row of the alphanumeric section, or the numeric section

you hold down a key for more than a moment	the corresponding character is entered repeatedly, until you release the key
you want to send a line to the system (a BASIC line, a PCOS command, or data in response to an INPUT/ LINE INPUT statement)	press CR, which positions the cursor at the beginning of the next line on the screen.
you want to move the cursor to a new line before reaching the margin	you must press SPACE the requisite number of times

CONTROL CHARACTERS

Control characters are entered when pressing either CTRL or COMMAND and another key together. The table below summarises all the M20 control characters.

IF you press...	THEN...
CTRL C (Break)	you interrupt program execution. M20 returns to BASIC Command Mode and displays Ok (if you are in BASIC) or to PCOS and displays > (if you are in PCOS). See also "Correcting Typing Errors" below
CTRL G	the cursor changes its shape and blink rate and the display of entered characters is suppressed (Hide State). TO RETURN TO NORMAL DISPLAY STATE YOU MUST PRESS CTRL G AGAIN, OR CR
CTRL H (Backspace)	the last character typed is deleted and the cursor is moved one position to the left
CTRL RESET (Logical Reset)	the memory is cleared and PCOS is loaded again from disk

CTRL S	screen output is suspended THE SUSPENDED OUTPUT IS RESUMED BY TYPING ANY KEY
CTRL HOME (Escape)	Insert State is exited, while remaining in Line Edit Mode (see Chapter 3)
COMMAND ?/	enables Shift Lock for letters (A-Z). TO DISABLE SHIFT LOCK, PRESS COMMAND ?/ AGAIN

CORRECTING TYPING ERRORS

You can correct typing errors either before or after you have sent a line to the system.

IF you notice an error	THEN...	OR...
before you have completed a line (i.e. before pressing CR)	delete the last character(s) by CTRL H and retype it/them	move the cursor to the next line by CTRL C and retype the line
after you have completed a line (i.e. after pressing CR), AND IF the line is a program line	enter the line correctly with the same line number THE NEW LINE WILL REPLACE THE OLD ONE	enter Line Edit Mode and use Line Edit Mode Commands (see Chapter 3)

USING THE M20 AS A CALCULATOR

You may use the M20 as a calculator for quick computation, and debugging purposes.

You are in BASIC. The special prompt Ok is on the screen.

You may enter P R I N T (or simply ?), followed by an expression and CR. The expression is evaluated and its value displayed. You may also enter L E T followed by a variable name (a string of characters whose first character is a letter), followed by an assignment operator (=), then by an expression, and CR. The expression is evaluated and its value assigned to the variable. You may use the variable to represent that value in successive computations.

The following table gives some examples.

DISPLAY	COMMENTS
PRINT 3 3 Ok	the constant 3 is displayed (a constant may be considered a simple expression)
PRINT 2+3 5 Ok	the expression 2+3 is evaluated, and its value (5) is displayed
LET A=15.21 Ok	the constant 15.21 is assigned to the variable A. You may use A in successive computations to represent this value
?A-1 14.21 Ok	the expression A-1 is evaluated, and its value (14.21) is displayed. Note: ? is equivalent to PRINT
B=2.3 Ok	the constant 2.3 is assigned to the variable B. The keyword LET is optional, you may begin with a variable name
?A*B 34.983 Ok	the expression A*B is evaluated. The symbol * means "multiplied by". Its value (34.983) is displayed
?A*B-4Ø -5.Ø17 Ok	the expression A*B-4Ø is evaluated, and its value (-5.Ø17) is displayed. Note: If a value is negative, the minus sign is displayed, if a value is positive, no sign is displayed

WHAT IS BASIC?

A SAMPLE PROGRAM

You may also use the M20 to enter and run BASIC programs.

By writing and running a program you may solve problems that could not be solved using the M20 as a calculator.

A BASIC program consists of a series of statements. A statement is a complete instruction in BASIC, telling the M20 to perform specific operations.

You may enter either one or several statements per line. In the latter case, each statement must be separated by a colon (:).

Each line in a BASIC program begins with a line number: an integer greater than or equal to Ø and less than or equal to 65529 and ends when you press CR.

You are in BASIC. The special prompt Ok is on the screen. A sample program may be constructed by entering the following statements:

```
1Ø REM RECTANGLE1 CR
2Ø INPUT "Length";L CR
3Ø IF L<=Ø THEN 2Ø CR
4Ø INPUT "Width";W CR
5Ø IF W<=Ø THEN 4Ø CR
6Ø LET AREA=L*W CR
7Ø PRINT "Area=";AREA;" L=";L;" W=";W CR
8Ø GOTO 2Ø CR
9Ø END CR
```

These statements form a complete program that solves a very simple problem.

The problem is to find the area of a rectangle by entering the values of length and width via the keyboard. It has been selected both for its simplicity and to illustrate a variety of BASIC features. Other more concise solutions exist (as we shall see in Chapter 3).

You have entered one statement per line. You could also enter more than one statement per line, using the colon (:) as statement separator and reducing the number of lines. For example:

```
1Ø REM RECTANGLE1 CR
2Ø INPUT "Length";L:IF L<=Ø THEN 2Ø CR
3Ø INPUT "Width";W:IF W<=Ø THEN 3Ø CR
4Ø LET AREA=L*W CR
5Ø PRINT "Area=";AREA;" L=";L;" W=";W CR
6Ø GOTO 2Ø:END CR
```

You may enter up to 255 characters per (logical) line, including the line number. A logical line may appear on the screen on several physical lines. For example:

```
2Ø INPUT "Length";L:IF L<=Ø
                    THEN 2Ø CR
```

is one logical line divided into two physical lines. To change lines before reaching the margin press the SPACE key the requisite number of times.

KEYWORDS

Each statement begins with a keyword (or reserved word). The keyword is a mnemonic of an English word. It must be preceded and followed by at least one blank.

Note: You may not use a keyword as a variable name.

The keyword defines the type of statement to be carried out. One or more operands (constants or variables) or expressions follow the keyword. Some statements have more than one keyword e.g. IF... THEN. The statements of our program contain the keywords REM, INPUT, IF... THEN, LET, PRINT, GOTO and END. BASIC keywords may be entered in lower case or upper case letters. They are converted to upper case letters when listing the program (see Chapter 2). Besides keywords, other reserved words are BASIC command names (e.g. RUN, LIST etc..) and function names (e.g. SIN, COS, etc.). See Appendices C,D, and E for a complete list.

WHAT IS BASIC?

CONSTANTS

Specific numbers (such as Ø, 15Ø, - 31.7) are called "numeric constants" and specific strings (such as "Length", "Width", "Area =", " L=" and " W=") are called "string constants". This means that their values remain the same throughout program execution. For example when the constant 15Ø is used in a program, it remains fixed at that value throughout program execution. Numeric costants may be integer (e.g. 15Ø) or non integer (real) e.g. - 31.7. String constants are always quoted (i.e. included in a pair of quotation marks), unless differently specified. Unquoted strings may be used for instance within DATA statements and answering to an INPUT or LINE INPUT statement. For further information see Chapter 4.

VARIABLES

A variable is a named data item whose value may change during program execution. The length of the name of a variable may be maximum of 40 characters. The first character must be a letter. Examples of variables in our program are:

L , W , AREA

Like keywords, variable names may be entered either in lower or upper case letters. They are converted to upper case letters when listing the program.

A variable may be a simple variable (e.g. L,W,AREA mentioned above) or a subscripted variable.

A subscripted variable is an array element, i.e. an element of a collection of variables under one name. You can distinguish different elements by the value(s) of one or more subscripts appearing in parentheses after the array name. For example, if A is a one dimensional array, A(Ø) is the first element, A(1) the second element, and so on.

An array may have any number of dimensions. A one dimensional array might be thought of as a list of items. A two dimensional array is like a table of values. In this case the first subscript designates the "row" in the array and the second subscript designates the "column", for example B(1,2) is the element belonging to second row and the third column.

For further information see Chapter 4.

FUNCTIONS

We can classify functions as either built-in or user-defined functions.

We shall speak briefly of built-in functions here, whilst user-defined functions will be described later (see Chapter 9, where you can also find detailed information on all BASIC functions).

Built-in functions provide a set of commonly used numeric operations (as square root, sine and natural logarithm etc...) and string operations (as extracting group of characters - a substring - from a larger string etc.). The user can invoke them within any BASIC program, writing the name of the function (e.g. SIN followed in parentheses by the value(s) of one or more arguments (e.g. SIN (1.5)).

A function call may be an operand within an expression (e.g. 1 + 2*COS(.4)) and the arguments may also be expressions (e.g. LOG(45/7)).

For example:

SIN(1.5) is .997495	(SIN returns the sine of the argument)
1 + 2*COS(.4) is 2.84212	(COS returns the cosine of the argument)
LOG(45/7) is 1.86Ø75	(LOG returns the natural logarithm of the argument)
SQR(1Ø) is 3.16228	(SQR returns the square root of the argument)

EXPRESSIONS

We can classify expressions as numeric, string, relational or logical.

Let us define briefly what we mean for numeric, string and relational expressions.

Logical expressions will be defined later (see Chapter 6, where we shall also describe all the types of expressions in detail).

Numeric Expressions

A numeric expression can be either a numeric constant, a simple numeric variable, a numeric array element, a numeric function, or a mixture of them linked by means of special symbols, called numeric operators.

The numeric operators are:

+ addition (e.g. A+B+C)

- subtraction (e.g. A-B)

\ integer division (the operands are rounded to the nearest integers before the division is performed, and the quotient is truncated to an integer, e.g. 25.68\6.99 is 3)

MOD modulus arithmetic (it gives the integer value which is the remainder of an integer division, e.g. 25.68 MOD 6.99 is 5, as 26/7 is 3 with remainder 5)

* multiplication (e.g. A*B)

/ division (e.g. A/B)

- (negation it changes the sign of the operand, e.g. -A is 35 if the value of A is -35)

^ exponentiation (e.g. A^B)

String Expressions

A string expression can be either a string constant, a simple string variable, a string array element, a string function, or a mixture of them linked by plus signs (+).

By using the plus sign, strings can be joined - "concatenated" is the technical term.

For example:

```
1Ø A$ = "Chicago,"
2Ø B$ = "IL.,"
3Ø C$ = A$+B$+"USA"
```

The concatenation in statement 3Ø would result in C$ being assigned the string:

```
Chicago,IL.,USA
```

Relational Expressions

Relational expressions compare either two numeric or two string expressions by means of a relational operator.

The relational operators are:

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

The result of a relational expression expression may be true or false and may be used to make a decision regarding program flow. For example we used a relational expression in the statement:

```
3Ø IF L<=Ø THEN 2Ø
```

It returns control to statement 2Ø if L is negative or zero.

THE USE OF BLANKS

Blank spaces may be inserted in the statements to make them more readable. The use of blanks is almost always optional in BASIC with the following exceptions:

- at least one blank must precede and follow a keyword

- blanks are significant within string constants

- blanks are forbidden within numeric constants (including line numbers), keywords, variable names, and function names.

For example:

```
2Ø INPUT "Length"; L
```

and

```
2Ø  INPUT  "Length"; L
```

are equivalent, but

```
2Ø INPUT "L e n g t h"; L
```

is not equivalent, as it contains a longer string constant.

COMMENTS

You may document your program by the REM (Remark) statement. After REM you may enter any string of printable ASCII characters. For example:

1Ø REM RECTANGLE1 **CR**

Another way of documenting your program is the through use of comment fields (a string of printable ASCII characters preceded by an apostrophe and ended by **CR**).

For example:

1ØØ GOTO 1ØØ 'Loops for ever **CR**

Both REM and comment fields may be inserted anywhere in your program as they are not executable statements but they appear on the program listing and increase the readability of your program. For further information see Chapter 2.

RUNNING OUR PROGRAM

Let us run our sample program. If you have already entered it via Keyboard (and have not switched the M20 off in the interim) it will be in memory. Enter **L** **I** **S** **T** **CR**; the listing will appear on the screen. At the end of the listing when Ok appears on the screen, enter **R** **U** **N** **CR**.

DISPLAY

```
LIST
1Ø REM RECTANGLE1
2Ø INPUT "Length";L
3Ø IF L<=Ø THEN 2Ø
4Ø INPUT "Width";W
5Ø IF W<=Ø THEN 4Ø
6Ø LET AREA=L*W
7Ø PRINT "Area=";AREA;" L=";L;" W=";W
8Ø GOTO 2Ø
9Ø END
Ok
RUN
Length? 3.5
Width? 4.2
 Area= 14.7  L= 3.5  W= 4.2
```

COMMENTS

M20 begins executing statements sequentially. Because statement 1Ø is a REM(ark) it is not executed; execution in this case starts with statement 2Ø.

When an INPUT statement is encountered (see statements 2Ø and 4Ø) program execution is suspended and M20 prompts a message indicating that you should enter a value. You could enter for example 3.5 for the length and 4.2 for the width.

```
Length? -7.3
Length? 7.3
Width? 1.3Q
?Redo from start
Width? 1.32
Area= 9.636  L= 7.3  W= 1.32
Length? ^C
Break in 2Ø
Ok
```

Statement 6Ø calculates the value of AREA. Statement 7Ø displays the values of AREA, L and W. Statement 8Ø returns control to statement 2Ø.

If you enter a negative value (e.g. -7.3), for L, statement 2Ø is executed again, as statement 3Ø returns control to statement 2Ø if L is negative or zero.

If you enter a negative value for W, statement 4Ø is executed again, as statement 5Ø returns control to statement 4Ø, if W is negative or zero.

If you enter a string value for L or W (e.g. 1.3Q for W) the M20 displays an error message:

?Redo from start

and you must re-enter the value. This program continues to run until you press CTRL C to stop execution. The M20 displays a break message and enters Command Mode. To resume execution, enter C O N T ↵R

WHAT IS BASIC?

MODES OF OPERATION

BASIC has three modes of operation.

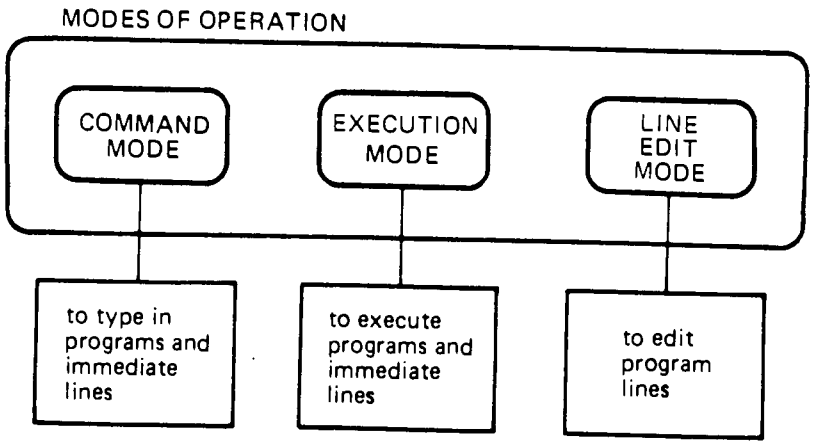


Figure 1-3 Modes of Operation

COMMAND MODE

Whenever the M20 enters Command Mode, it displays a special prompt:

Ok

In Command Mode, BASIC does not accept your input until you complete the line by pressing **CR**.

Program and Immediate Lines

BASIC always ignores leading spaces in a line - it jumps ahead to the first non-space character. If this character is not a digit, BASIC treats the line as an immediate line. If it is a digit, BASIC treats the line as a program line (see below).

IF...	THEN...
you enter a program line, i.e. a line number (Ø to 65529), one or more BASIC statements or commands (separated by colons) and CR	the line is stored in memory, when you press CR. The line is not executed until you enter R U N CR. For example: 1ØØ PRINT "The LOG of 5 is";LOG(5) is a program line. When you press CR BASIC stores it in memory. To execute it, press R U N CR
you enter an immediate (direct) line, i.e. one or more BASIC statements or commands (separated by colons) and CR	the line is executed as soon as you press CR. For example: PRINT "The LOG of 5 is";LOG(5) is an immediate line. When you press CR BASIC executes it
you enter a sequence of program lines	the lines are stored in memory to form a BASIC program. They are stored in line number sequence, irrespective of the order they were entered. The program is not executed until you enter R U N CR

Submodes

Command Mode includes the following Submodes:

- Immediate (or Direct), when you enter an immediate line

- Program, when you enter a program line.

WHAT IS BASIC?

EXECUTION MODE

The M20 executes both BASIC immediate and program lines in BASIC Execution Mode and PCOS commands in PCOS Execution Mode. A BASIC program is executed in ascending line number sequence, unless a control statement (GOTO, ON...GOTO, IF...THEN...ELSE, IF...GOTO...ELSE, FOR/NEXT, WHILE/WEND) dictates otherwise.

LINE EDIT MODE

BASIC includes a Line Editor for correcting program lines. This is useful for correcting long and complicated lines without having to re-enter them completely.

IF you wish to edit	THEN you must enter...	BASIC displays
the current line (line-number 11...1)	EDIT SPACE . CR	11...1
a specified line (line-number nn...n)	EDIT SPACE n n...n CR	nn...n

Note: The current line is the last line entered or edited. If while running a program an error is encountered the line containing the error becomes the current line. (See "Syntax Errors" below).

If M20 enters Edit Mode, you can begin editing on the line (deleting, inserting, and replacing characters) by use of Edit Mode subcommands (see Chapter 3).

In Edit Mode BASIC takes your input as soon as you enter a character, without waiting for you to press CR. By pressing CR BASIC exits Edit Mode.

Syntax Errors

If, during execution of a program line, a syntax error is detected, M20 displays:

```
Syntax error in nn...n
Ok
nn...n
```

and automatically enters the Line Edit Mode.

Here nn...n stands for the line number where the error occurred.

Edit States

Line Edit Mode provides the following states:

- Delete

- Change

- Insert

To enter these states or to exit from them, you must use the appropriate Edit Commands (see Chapter 3).

BASIC STATEMENTS AND COMMANDS

It is sometimes difficult to distinguish a BASIC statement from a BASIC command, as both may be used in a program or an immediate line, but:

- BASIC statements are generally used in program lines and entered in sequence to form a program (with the exception of PRINT, LPRINT, LET and SWAP, which are also often used in immediate lines, when using the M20 as a calculator or for debugging purposes)

- BASIC commands are used to manipulate programs and for utility purposes, such as listing programs or clearing the memory. They are generally used in immediate lines (with the exception of KILL, LOAD, RUN, SYSTEM, TROFF, TRON and WIDTH which are also often used in a program).

Later in the manual, when introducing a BASIC statement or command we will write:

- (IMMEDIATE), if it may be used only in an immediate line

- (PROGRAM), if it may be used only in a program line

- (PROGRAM/IMMEDIATE), if it may be used both in a program and an immediate line.

Instead we shall not write anything after the name of a function, as functions may be used both in a program and immediate line.

CHANGING MODE OR ENVIRONMENT

The operation mode (Command, Edit, or Execution Mode) or environment (PCOS or BASIC) may be changed by entering certain commands or control characters, or if certain conditions occur.

The table below summarizes how you can change mode or environment.

IF M20 is in...	AND IF...	THEN...
BASIC Execution Mode	you press: **CTRL** **C**	execution is interrupted and M20 enters BASIC Command Mode
	when M20 is executing a BASIC program or an immediate line you press: **CTRL** **RESET**	memory is cleared and PCOS is reloaded
	a Syntax error is detected	M20 enters BASIC Line Edit Mode at the line that caused the error
	the execution of a BASIC program or command is completed OR an error other than a syntax error is detected OR a STOP (or END) statement is encountered	M20 enters BASIC Command Mode

BASIC Command Mode	you enter an immediate line	M20 enters BASIC Execution Mode, executes the line and returns to BASIC Command Mode
	you enter: **S** **Y** **S** **T** **E** **M** **CR** Note: SYSTEM may also be used in a BASIC program	M20 enters PCOS. Both the BASIC interpreter and the user memory are cleared.
	you enter: **E** **D** **I** **T** **SPACE** **n** **n**...**n** **CR** OR **E** **D** **I** **T** **SPACE** **.** **CR**	M20 enters BASIC Line Edit Mode
	you press: **CTRL** **RESET**	memory is cleared and PCOS is reloaded
BASIC Line Edit Mode	you press: **CR**	M20 enters BASIC Command Mode. (The newly modified line is displayed). All program variables are cleared
	you press: **E** (Exit)	M20 enters BASIC Command Mode. The remainder of the newly modified line is not displayed and all program variables are cleared
	you press: **Q** (Quit Editing)	M20 enters BASIC Command Mode and cancels all the changes that were made to the line. No program variable is cleared

WHAT IS BASIC?

	you press: CTRL RESET	memory is cleared and PCOS is reloaded
PCOS	you enter: B A CR	BASIC is loaded and M20 enters BASIC Command Mode
	you enter any other PCOS command	M20 enters PCOS Execution Mode
	you press: CTRL RESET	memory is cleared and PCOS is reloaded
	you enter a file identifier that has the BAS extension (e.g. FILEA.BAS). OR you enter the BASIC command followed by a file identifier (e.g. BA FILEB)	M20 enters BASIC Execution Mode and the specified program is executed

2. ENTERING, LISTING, AND EXECUTING A PROGRAM

ABOUT THIS CHAPTER

This chapter illustrates notation convention, how to document a program, and the most useful BASIC commands. These allow you to enter, list, save, load and execute programs.

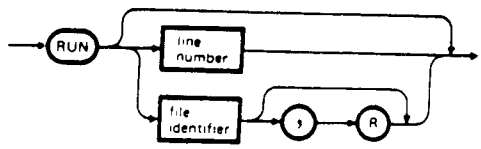
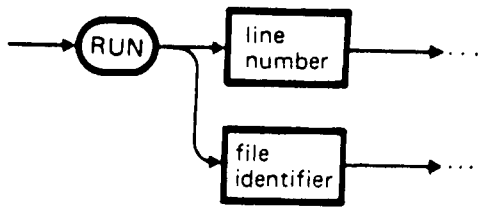
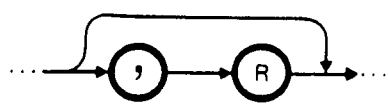
CONTENTS

ENTERING, LISTING, AND EXECUTING A PROGRAM

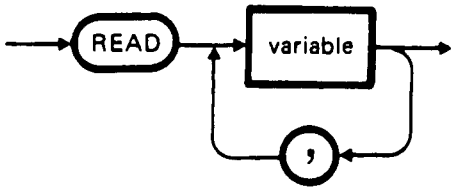
NOTATION CONVENTION

The syntax of BASIC is described by means of syntax diagrams.

A syntax diagram is a flow-chart with one entry and one exit. Each path through the diagram defines an allowable sequence of symbols. The following table summarizes the rules the user must follow to draw a syntax diagram.

NO.	RULE	EXAMPLE
1	all items enclosed by a rounded envelope (ovals or circles) must be entered exactly as shown. Items enclosed in a rectangular box are names of parameters used in a statement, a command, or a function. A description of each parameter is given in the text following the drawing	
2	a fork indicates a choice: you must select one path. For example after RUN you may either: - enter a line number OR - a file identifier	
3	a branch without a parameter indicates that the alternative is a bypass (used to indicate ,R is optional)	

4 a loop indicates a repetition. For example variable may be repeated n times in a READ statement, and each variable is separated from the next one by a comma



5 this manual shows BASIC reserved words in upper case letters, even though you may enter them in lower case letters. Some examples of reserved words are shown on your right. They are the keywords of our sample program (RECTANGLE1)

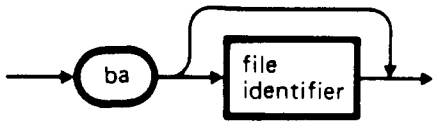
```
REM
INPUT
IF...THEN
LET
PRINT
GOTO
END
 .
 :
```

6 PCOS command names are mnemonic. For example:

BASIC - go to BASIC
VCOPY - volume copy

and so on.

They may be entered either in lower case or in upper case letters. They are normally entered in lower case letters and in their short form (the first two letters) as shown in the syntax diagrams



DOCUMENTING A PROGRAM

Often you may want to insert comments in order to make your program logic easier to follow. This can be done by using:

- the REM statement, or

- comment fields.

REM/COMMENT FIELDS (PROGRAM)

The REM (Remark) statement is one way to document your program. You can write any message you want following the keyword REM.

Another way to document your program is to write a comment field, i.e. a string of characters preceded by an apostrophe (') and ended by CR.

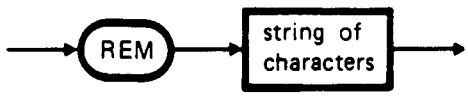


Figure 2-1 REM Statement

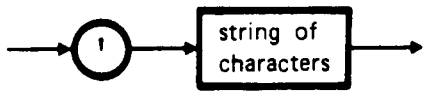


Figure 2-2 Comment Field

Examples

IF you enter...	THEN...
1Ø REM RECTANGLE1 CR	a REM statement titles your program. It is good programming practice to title programs
1ØØ REM SUBROUTINE1 CR	a REM statement marks the beginning of a subroutine (see Chapter 1Ø). It is good programming practice to title subroutines.

1Ø 'RECTANGLE1 CR	a comment field titles your program. Note: In this case the apostrophe works like REM, as the comment occupies a line
15Ø LET A=(A1+A2)/2 'Average CR	a comment field ends a statement

Remarks

A REM statement may not be followed by other statements on the same line. It can however be the last statement on a multi statement line.

A comment field may:

- occupy a line (in which case the apostrophe has the same function as REM)

- end a statement.

Both REM and comment fields can appear anywhere in a program. They are not executable statements, but they appear in the listing.

ENTERING A PROGRAM

The Ok prompt is on the screen. BASIC is waiting for you to start. You might start immediately by entering the first statement, beginning by entering the line number 1Ø. There is, however, a preliminary step that can make your task a little easier. You can request the system to number your lines for you. You do this with the AUTO command, which is described below.

On the other hand, if you have already entered a program and you want to enter a new one, you must first enter a NEW command. This causes the program in memory to be deleted allowing you to enter a new program (see below). The program in memory is also deleted when LOADing a new program from disk, or when entering a SYSTEM command (to return to PCOS), or when turning off the machine.

In these cases it would be wise to save your program first (unless you already have a copy).

Let us enter our sample program (RECTANGLE1), by pressing the following keys:

N **E** **W** **SPACE** **CR**

This will clear the memory.

Then enter:

```
1Ø REM RECTANGLE1 CR
2Ø INPUT "Length";L CR
3Ø IF L<=Ø THEN 2Ø CR
4Ø INPUT "Width"; W CR
5Ø IF W<=Ø THEN 4Ø CR
6Ø LET AREA = L*W CR
7Ø PRINT "Area="; AREA; " L=";L;" W=";W CR
8Ø GOTO 2Ø CR
9Ø END CR
```

It is conventional to use an interval of 1Ø between each line number. This allows you to modify a program simply by inserting statements between existing lines.

Although program lines can be entered in any order they are ordered in memory in ascending line number sequence.

For example, we may enter the statement whose line number is 5Ø, then the statement whose line number is 1Ø etc... and obtain the same listing (i.e. the same program).

You may enter keywords and variable names in upper case or lower case letters. They will be converted into the corresponding upper case letters when listing the program.

AUTO (IMMEDIATE)

Starts automatic line numbering.

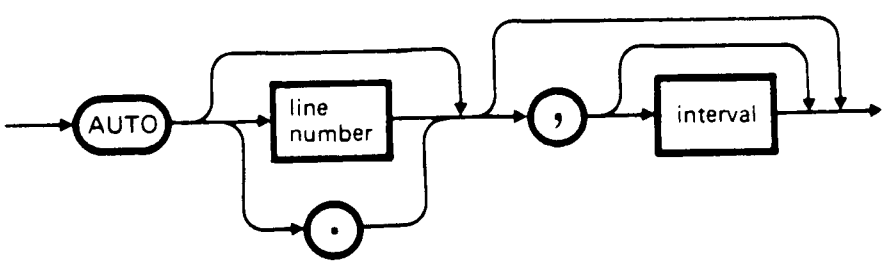


Figure 2-3 AUTO Command

Where

SYNTAX ELEMENT	MEANING	DEFAULT VALUES
line number	the first line number generated	1Ø (if the interval is omitted, otherwise Ø)
	the first line number generated is the number of the current line	
interval	is the interval between line numbers	1Ø

Examples

IF you enter...	THEN line numbering begins...	AND line interval is...
AUTO CR	at line 1Ø (default value)	1Ø (default value)
AUTO .,3Ø CR	at the current line	3Ø
AUTO 1ØØ CR	at line 1ØØ	1Ø (default value)
AUTO 15Ø, CR	at line 15Ø	the last interval specified by a preceding AUTO command or, if none were preceding, 1Ø (the default value)
AUTO 2ØØ,2Ø CR	at line 2ØØ	2Ø
AUTO, 3 CR	at line Ø	3

An Asterisk after Line Number

IF...	THEN...
AUTO generates a line number that already exists	an asterisk is displayed after the line number to warn the user that any input will replace an existing line. However, typing **CR** immediately after the asterisk will save the existing line and generate the next line number. Note: This will happen only if you enter AUTO when a program already exists

To Terminate Line Numbering

IF you press...	THEN...
CTRL **C**	M20 terminates automatic line numbering and Command Mode is entered. Note: The line in which **CTRL** **C** is pressed is not saved

NEW (PROGRAM/IMMEDIATE)

Deletes the current program and variables allowing you to enter a new program.

NEW switches off the trace flag in the same way as TROFF (see Chapter 13) and closes all data files (see Chapter 12).

Figure 2-4 NEW Command

Examples

IF you enter...	THEN...
NEW **CR**	the program currently in memory is deleted
1Ø REM RECTANGLE1 **CR** 2Ø INPUT "Length";L **CR** . . .	you enter a new program from keyboard.
	Note: It is not necessary to enter NEW before loading a program from disk, by issuing a LOAD or a RUN command (as they automatically clear memory)

LISTING A PROGRAM

Once a program is in main memory it can be listed. To list your program, enter either the LIST command (the listing will appear on the screen) or the LLIST command (the listing will appear on the printer).

You cannot list a protected program (SAVEd with the P option, see below). The LIST and LLIST commands edit your programs by converting to upper case letters any reserved word (keyword, variable names, and function names) and to PRINT any question mark (?) used instead of PRINT. Moreover statements are ordered in ascending line number sequence, even though you may have entered them in a different order.

To list our sample program on the screen enter **L** **I** **S** **T** **CR**. You will see the following.

```
LIST
1Ø REM RECTANGLE1
2Ø INPUT "Length";L
3Ø IF L<=Ø THEN 2Ø
4Ø INPUT "Width";W
5Ø IF W<=Ø THEN 4Ø
6Ø LET AREA=L*W
7Ø PRINT "Area=";AREA;" L=";L;" W=";W
8Ø GOTO 2Ø
9Ø END
Ok
```

At the end of a listing the M20 enters Command Mode and displays Ok.

LIST/LLIST (IMMEDIATE)

LIST lists program lines on the screen, LLIST lists program lines on the printer.

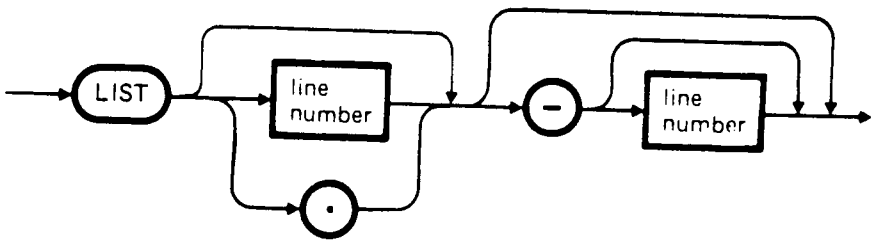


Figure 2-5 LIST Command

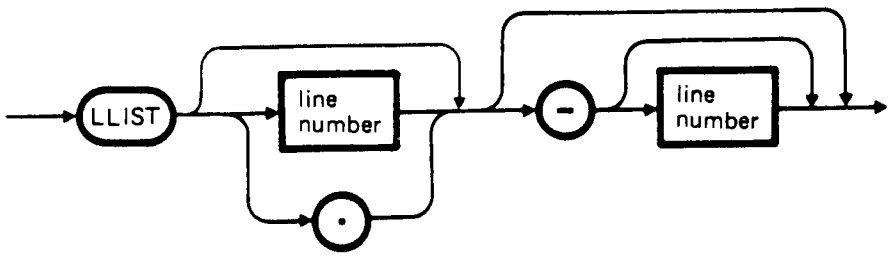


Figure 2-6 LLIST Command

Examples

IF you enter...	THEN...
LIST CR	the entire program is listed
LIST 15Ø CR	line 15Ø is listed
LIST . CR	the current line is listed
LIST 2ØØ- CR	line 2ØØ and all higher-numbered lines are listed
LIST -1ØØØ CR	all lines from the beginning to 1ØØØ are listed
LIST 1ØØ-19Ø CR	all lines from 1ØØ to 19Ø are listed
LIST .-5ØØ CR	all lines from the current line to 5ØØ are listed

Suspending a Listing

IF...	THEN...
you press: **CTRL** **S**	listing is suspended, without entering Command Mode. You may continue the suspended listing by typing any key
you press: **CTRL** **C**	M20 enters Command Mode and abandons the listing
the end of the program is reached	listing is terminated and Command Mode is entered

PROGRAM AND DATA FILES

A file is a sequence of statements (program file) or data (data file) which may be stored on a disk.

The table below sumarizes the main characteristics of program and data files.

FILE TYPE	MEANING
program files	a program file is a sequence of program lines. They are stored in memory in line number sequence, irrespective of the order in which they were entered. A program file is stored in memory in a packed binary format, and saved on a disk either in this format or in ASCII format (if you use the A option to save it). ASCII format files are sequences of ASCII characters; effectively they contain the source listing of your program. When loaded into memory (by a LOAD or RUN command) they are always converted into packed binary format

data files

a data file is a sequence of numeric and/or string data, which is stored on a disk.

A data file is created by a BASIC program. First of all it must be opened by an OPEN statement which specifies the access mode, a file number and the name of the file. The value of the file number must be in the range 1 to 15.

Each following Input/Output statement in the program will specify the file by the file number.

When you have finished with the file, it is good programming practice to "close" it using the CLOSE statement. In any case all data files will be closed when an END statement is encountered.

Note: When closing a data file, the program cannot access it unless a new OPEN statement is executed. This may specify a new file number and a new access mode. Only the file name must remain the same

FILE AND VOLUME IDENTIFIERS

A disk may contain one or more program and/or data files. A single file may not be fragmented over more than one disk.

A group of files stored on the same disk forms a "volume". Each file and each volume has an identifier. Each file name must be unique on any one volume. Saving a program file which already exists on a volume causes the original file to be overwritten.

You may assign an identifier to a file either by an OPEN statement (data files), or by a SAVE command (program files), or by a FNEW PCOS command. You may assign an identifier to a volume by a VFORMAT, a VNEW, or a VRENAME PCOS command.

The system recognizes a volume identifier and can find any of its files only if the corresponding diskette is mounted in a drive. This restriction will not be applied to the hard disk, as this unit is always on line.

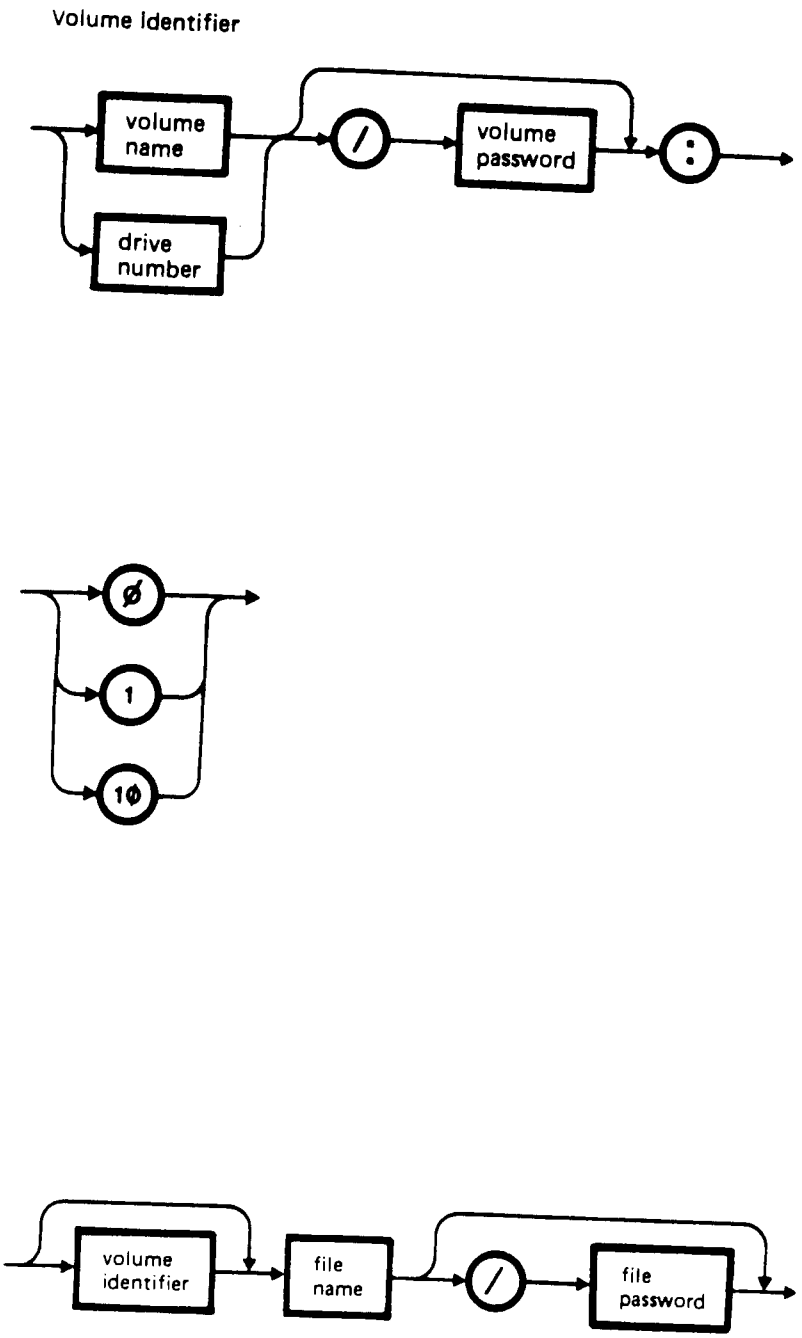


Figure 2-7 File and Volume Identifier

Where

SYNTAX ELEMENT	MEANING
volume name	string of up to 14 printable ASCII characters (for illegal characters see below). To select a volume in a PCOS or BASIC command or in an OPEN statement you must specify a volume name or a drive number. The volume name (or the drive number) may be followed by a volume password. At the end of a volume identifier a colon must be entered. For example: SAVE "VOL1:FILE1" CR Here VOL1 is a volume name, FILE1 is a file name and VOL1:FILE1 is a file identifier. You save the program file FILE1 on the disk named VOL1 (for more details see the SAVE command below) Note: When specifying a file or volume identifier in a BASIC statement or command you must either include the identifier in a pair of quotation marks, or write a string variable or a string expression whose value is the identifier. When specifying a file or volume by name in a PCOS command you must not include the identifier in a pair of quotation marks. For example: vn VOL1: CR Note: In BASIC a volume identifier may be specified only if included in a file identifier. The only exception is with the FILES command when you want to list all the files of a volume. For example: FILES "VOL2:" CR
drive number	the drive number may be either Ø (indicating the drive on the right), or 1 (indicating the drive on the left) or 1Ø (indicating the hard disk drive). With a hard disk system drive Ø is on the left and drive 1 does not exist. For example:

LOAD "1:FILEØØ2" CR

Here 1: indicates that file FILEØØ2 resides on the diskette inserted in drive 1. The command loads the file into memory (for more details see LOAD command below)

file password
OR
volume password

string of up to 14 printable ASCII characters (for illegal characters see below).

Passwords give the user protection at volume or file level (see below). They may be entered after a volume name, a drive number, or a file name and are preceded by a slash. For example:

RUN "Ø:RECTANGLE1/R1" CR

Here you load file RECTANGLE1 which has the password R1 and run it. RECTANGLE1 resides on the diskette inserted in drive Ø (for more details see the RUN command below)

file name

string of up to 14 printable ASCII characters (for illegal characters see below).

To select a file in a PCOS or BASIC command or in an OPEN statement you must specify a file name. The file name may be preceded by a volume identifier and followed by an extension and/or by a (file) password. For example:

SAVE "1:PRIMENUMBERS/PN" CR

Here you save the BASIC program PRIMENUMBERS on the diskette inserted in drive 1 and give it the password PN.

Note: If you do not specify any volume identifier before the file name,the search is limited to the last selected drive.

Note: The file name may include an extension name, i.e. a string of up to 12 printable ASCII characters, preceded by a period (.). For illegal characters see below.

Note: filename.extension cannot exceed 14 characters in total (including only one period). For example:

LOAD "FILEA.CHAR" CR

will load FILEA which has the extension CHAR. It resides on the last selected drive.

Note: Some extensions have special meanings: BAS (BASIC programs); CMD (PCOS transient commands); SAV (PCOS transient commands which become resident the first time they are executed). For more details see "Professional Computer Operating System (PCOS) User Guide".

Illegal Characters

comma (,)	pound (#)	slash (/)	backslash (\)
plus (+)	ampersand (&)	colon (:)	single quote (')
asterisk (*)	greater than (>)	dollar ($)	question mark (?)
quote (")	equals (=)	at sign (@)	exclamation (!)
hyphen (-)	semicolon (;)	space	

or any control character

PASSWORDS

Passwords give the user protection at volume or file-level as desired.

If a password has been assigned to a volume it must be specified to enable the volume. By convention a volume is said to be enabled either if it has no password or if the password has been specified in a BASIC or PCOS command.

The user must enter the corresponding password correctly on all occasions when using volume and file identifiers.

Note: If you have forgotten a password for a file or volume you will not be allowed access to that file or volume by BASIC or PCOS.

VOLUME PASSWORD

IF you want to...	THEN...
assign a password to a volume	issue a VPASS command, specifying the password. For example: vp MYVOL:,MYPASS **CR** IF the volume already has a password this must be specified by the VPASS command, which, in this case, will change the password. For example: vp VOL1/OLDPASS:,NEWPASS **CR**
access a volume that has a password (or a file saved on a volume that has a password)	enable that volume specifying the volume password after the volume name or the drive number, in a BASIC or PCOS command or in an OPEN statement. Note: Once a diskette password has been specified, it need not be specified again until the diskette has been removed and another diskette has been referenced in the drive unit in which the diskette was inserted. For the hard disk, once the password has been specified, it need not be specified again until PCOS is rebooted.
remove a volume password	issue a VDEPASS command. Note: You must know the password to use a VDEPASS command
hide a volume password	press **CTRL** **G**. The cursor will change its shape and blink rate and the display of entered characters is suppressed, (Hide State). To return to normal Display State you must press **CTRL** **G** again, or **CR**

FILE PASSWORD

IF you want to...	THEN...
assign a password to a file	issue an FPASS command, specifying the password IF the file already has a password, this must be specified by the FPASS command which, in this case, will change the password
assing a password to a program file (that has none)	and FPASS can be issued, or else the password can be specified in a SAVE command. For example: SAVE "FILEABC/PASSABC" CR
access a file that has a password	specify that password after the filename. For example: LOAD "FILEZ1/PASSZ1" CR If the volume also has a password, you must specify it too (unless the volume has already been enabled)
remove a file password	issue a FDEPASS command. Note: You need to know the file password to remove or change it
hide a file password	press CTRL G. The cursor will change its shape and blink rate and the display of entered characters is suppressed, (Hide State). To return to normal Display State you must press CTRL G again, or CR

WRITE PROTECTION

Write protection can be applied by the user at volume or file level.

IF you want to...	THEN...
write protect a volume (i.e. to prevent any writing to that diskette)	cover the write protect notch with an aluminized label Note: It is not possible to write protect the hard disk, but it is possible to write protect its files
unprotect a volume	remove the aluminized label
write protect a file	issue a FWPROT command, specifying the file identifier
unprotect a file	issue a FUNPROT command, specifying the file identifier

SAVING A PROGRAM

A program is kept in memory only as long as the M20 is switched on. As soon as you turn off the machine, your program is lost. If you want to retain your newly written program for future use, then you must issue a SAVE command to store it on a disk.

You can save the current program on other occasions too. The table below summarizes them. In any case the disk must be enabled otherwise you must specify the volume password in the SAVE command. Moreover, if you want to save the program on a diskette, this must not be write protected.

IF you want to...	THEN...
turn off the machine	save the current program (unless you already have a copy)
enter another program from keyboard	save the current program (unless you already have a copy)

load another program from disk (by entering a LOAD or RUN command)	save the current program (unless you already have a copy)
go to PCOS (by entering a SYSTEM command)	save the current program (unless you already have a copy)
replace the old version of your program	save the current program, specifying the same name as the old version AND the same password if the old version already has a password
save the current program in ASCII format	specify the A option in the SAVE command
protect the current program against any attempt to list, edit, or save it again	specify the P option in the SAVE command

Note: During a saving operation the disk-unit red light comes on. When it goes off, your program has been saved, and Ok appears on the screen.

SAVE (PROGRAM/IMMEDIATE)

Saves the current program on a disk, gives it a name, and optionally a password.

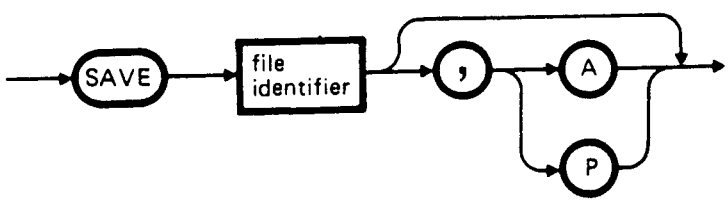


Figure 2-8 SAVE Command

Where

SYNTAX ELEMENT	MEANING
file identifier	may be either a string constant or a string variable. Specifies the name of the program to be saved. The file identifier may include a file password and a volume identifier
A	specifies that the program must be saved in ASCII format
P	specifies that the program must be saved protected against any attempt to list, edit, or save it again

Examples

In each of the following cases the volume must be enabled and must not be write protected.

IF you enter...	THEN...
SAVE "RECTANGLE1" CR	RECTANGLE1 is saved on the disk inserted in the last selected drive. RECTANGLE1 has no password
SAVE "Ø:RECTANGLE1" CR	RECTANGLE1 is saved on the diskette inserted in drive Ø. RECTANGLE1 has no password
SAVE "1Ø:RECTANGLE1" CR	RECTANGLE1 is saved on the hard disk. RECTANGLE1 has no password
SAVE "VOL1:RECTANGLE1" CR	RECTANGLE1 is saved on VOL1, which may be inserted in either of the two drives (as the volume name is specified). RECTANGLE1 has no password

SAVE "VOL1:R1/PASS" CR	R1 is saved on VOL1, which may be inserted in either of the two drives and assigns it a password

Replacing a File

In each case the volume must be enabled and must not be write protected.

IF you enter...	AND IF...	THEN...
SAVE "FILE1"	FILE1 already exists on the selected disk, AND has no password	the current program will replace the old version with the same name
SAVE "FILE1/PASS1"	FILE1 already exists on the selected disk, AND has the password PASS1	the current program will replace the old version with the same name and the same password
	FILE1 already exists on the selected disk, AND has a different password	no replacement takes place, and the system displays an error message (see Appendix C).
	FILE1 already exists on the selected disk, AND has no password	the current program will replace the old version with the same name and the new version will have the password PASS1

Option A

If you specify the A option, the file is saved in ASCII format.

If you do not specify the A option (i.e. either no option or the P option is selected), the file is saved in a packed binary format.

ASCII format takes more space on the disk than the packed binary format, but some commands require that files be in ASCII format. For instance the MERGE command requires an ASCII format file.

If you want to save a file using the "A" option, the maximum number of characters in a (logical) line is 255.

After BASIC executes a SAVE command with the "A" option in a program, it terminates.

IF you enter...	THEN...
SAVE "GEOMETRY",A CR	GEOMETRY is saved in ASCII format (i.e. a sequence of ASCII characters) on the disk inserted in the last selected drive. GEOMETRY has no password. The disk is presumed to be enabled.

Option P

If you specify the P option the file is not only saved in packed binary format, but it is also protected against any attempt to:

- list

- edit

- save it again.

Note: P protection cannot be removed.

IF you enter...	THEN...
SAVE "Ø:GEODESY",P CR	GEODESY is saved protected on the enabled diskette inserted in drive Ø. GEODESY has no password

LOADING A PROGRAM

If the program you want to enter into memory resides on a disk, you must issue a LOAD command.

LOAD deletes all variables and program lines currently residing in memory, thus before entering a LOAD command you should save the current program if you want to use it again. You do not have to save the current program if you already have a copy of it on disk.

To LOAD a program file from a disk, it must be enabled or you must specify the volume password in the LOAD command. To LOAD a program file which has a password, you must specify this file password in the LOAD command.

If you specify the R option all open data files are kept open, and the program is RUN after it is LOADed.

LOAD (PROGRAM/IMMEDIATE)

Loads a program file and optionally runs it.

Figure 2-9 LOAD Command

Where

SYNTAX ELEMENT	MEANING
file identifier	may be either a string constant or a string variable which specifies the program file to be loaded into memory from disk
R	specifies that all open data files are kept open and the program is RUN after it is LOADed

Examples

IF you enter...	THEN...
LOAD "1Ø:RECTANGLE1" CR	RECTANGLE1 is loaded from the hard disk. RECTANGLE1 has no password and its volume has already been enabled
LOAD "VOL1:RECTANGLE1/P1" CR	RECTANGLE1 is loaded from the volume VOL1 which may reside in either of the two drives. RECTANGLE1 has the password P1 and VOL1 is presumed to be enabled
LOAD "V3/P3:FAA" CR	FAA is loaded from the volume V3 which has the password P3. The volume V3 may be inserted in either of the two drives. Its password is indicated in the LOAD command to enable the volume. FAA has no password
LOAD B$	the program specified by the contents of the variable B$ is loaded into memory

Option R

If you specify the R option, all open data files are kept open and the program is RUN after it is LOADed.

If you do not specify the R option, LOAD closes all open data files.

Note that:

LOAD file identifier,R CR

and

RUN file identifier,R CR

have the same effect.

IF you enter...	THEN...
LOAD "ACCOUNT",R CR	program ACCOUNT is RUN after it is LOADed, and all open data files are kept open. ACCOUNT resides on the disk inserted in the last selected drive. ACCOUNT has no password and its volume has already been enabled

EXECUTING A PROGRAM

Once a program is in main memory, it can be executed (or "run", as this is frequently called). To tell the M20 to execute a program, you issue a RUN command (or a LOAD with the option R).

The RUN command runs the current program i.e. the program currently in memory; or loads a program from a disk and runs it. When the RUN command specifies a file identifier, this must include:

- the file password, if the file has a password
- the volume password, if the volume has a password (and it has not yet been enabled).

If you specify the R option all open data files are kept open.

Before entering a RUN file identifier (or RUN file identifier,R), save your current program (unless you already have a copy).

BASIC statements are executed in line number sequence, unless a control statement (GOTO, ON...GOTO, IF...GOTO...ELSE, IF...THEN.. ELSE, FOR/NEXT, WHILE/WEND) or a subroutine call statement (GOSUB, ON...GOSUB) dictates otherwise.

RUN (PROGRAM/IMMEDIATE)

Runs the program currently in memory or loads a program from disk and runs it.

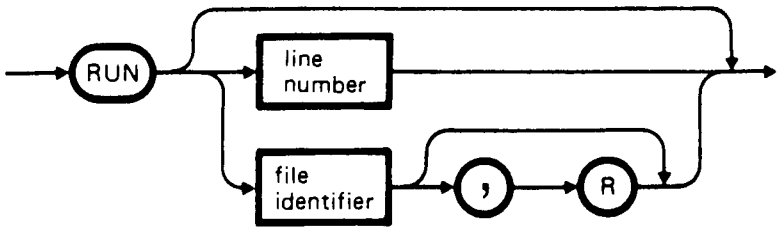


Figure 2-10 RUN Command

Where

SYNTAX ELEMENT	MEANING
line number	specifies the entry point of the program, i.e. the current program is run starting from the specified line number. If you do not specify a line number the current program is run from the beginning. Note: RUN line number and GOTO line number have the same effect, except that RUN also clears program variables
file identifier	may be either a string constant or a string variable which specifies the program file to be loaded from disk into memory and run
R	specifies that all open data files are kept open. If R is omitted all data files are automatically closed

Examples

In the following cases the volume is presumed to be enabled.

IF you enter...	THEN...
RUN CR	the current program is run
RUN 15Ø CR	the current program is run starting from line 15Ø
RUN "1:Newfile" CR	program Newfile is loaded into memory and run. It resides on the diskette inserted in drive 1, and has no password
RUN "NewVOL:Newfile" CR	program Newfile is loaded into memory and run. It resides on the disk named NewVOL which may be inserted in either of the two drives. Newfile has no password
RUN "1:Newfile/NewPASS" CR	program Newfile is loaded into memory and run. It resides on the diskette inserted in drive 1 and has the password NewPASS
RUN A$ CR	the program specified by the contents of the variable A$ is loaded into memory and run

Option R

If you specify the R option, all open data files are kept open.

If you do not specify the R option, RUN closes all open data files.

Note that:

RUN file identifier,R CR
and

LOAD file identifier,R CR

have the same effect.

IF you enter...	THEN...
RUN "1Ø:Newfile",R CR	program Newfile is loaded into memory and run, leaving the opened files open. Newfile resides on the hard disk. Newfile has no password and the hard disk is presumed to be enabled

Suspending Program Execution

IF...	THEN...
you press: CTRL C OR a STOP statement is encountered	a program interrupt occurs, the message "Break in line nnnnn" is issued and Command Mode is entered. No open file is closed. You can display program variables (by an immediate PRINT) or change their values (by an immediate LET). You can resume execution by entering a CONT command (unless you modify some statements).
an error is detected (except Syntax errors)	a program interrupt occurs, the error message is issued and Command Mode is entered. You cannot resume execution. No open file is closed; you can display program variables (by an immediate PRINT)
a Syntax error is detected	a program interrupt occurs, the "Syntax error" message is issued, and Edit Mode is entered at the line that caused the error. You can modify the line, but you cannot display program variables (unless you enter Command Mode by pressing Q). You cannot resume execution.

an END statement is encountered	a program interrupt occurs and Command Mode is entered. All open files are closed. You can display program variables (by an immediate PRINT) You can resume execution by entering a CONT command

Suspending Screen Output

IF...	THEN...
you press CTRL S	screen output is suspended, but no program interrupt occurs. No open file is closed. You cannot display program variables You can resume screen output by pressing any key.

3. UPDATING AND MODIFYING A PROGRAM

ABOUT THIS CHAPTER

Even an experienced programmer often needs to make changes and corrections to a program.

Your program can be updated in several ways e.g., deleting lines, replacing lines, inserting lines, renumbering lines, editing lines using the Line Editor.

This chapter describes these functions, making use of the sample program RECTANGLE1. Moreover it will explain how to rename a file, how to delete it from a disk, how to MERGE two programs and how to list the names of files residing on a specified disk.

Note that any modifications to the resident program will close data files and clear program variables.

CONTENTS

UPDATING AND MODIFYING A PROGRAM

DELETING LINES

We will use the program called RECTANGLE1 from chapter 2 as an example for demonstration purposes.

First of all, once the program RECTANGLE1 is in memory, issue a LIST command.

DISPLAY	COMMENTS
LIST 1Ø REM RECTANGLE1 2Ø INPUT "Length";L 3Ø IF L<= Ø THEN 2Ø 4Ø INPUT "Width";W 5Ø IF W<= Ø THEN 4Ø 6Ø LET AREA=L*W 7Ø PRINT "Area=";AREA;" L=";L;" W=";W 8Ø GOTO 2Ø 9Ø END Ok	RECTANGLE1 uses two separate INPUT statements for L and W. Let us modify the program to use only one statement. First delete line 4Ø

If you want to delete line 4Ø, enter:

D **E** **L** **E** **T** **E** **SPACE** **4** **Ø** **CR**

or

4 **Ø** **CR**

To see the result of this, issue another LIST command.

DISPLAY	COMMENTS
LIST 1Ø REM RECTANGLE1 2Ø INPUT "Length";L 3Ø IF L<= Ø THEN 2Ø 5Ø IF W<= Ø THEN 4Ø 6Ø LET AREA=L*W 7Ø PRINT "Area=";AREA;" L=";L;" W=";W 8Ø GOTO 2Ø 9Ø END Ok	As it stands now, RECTANGLE1 will not execute. You must now correct line 2Ø (which asks for only one input value) and line 5Ø (which refers to a line no longer in the program). We shall correct our program in the following pages

DELETE (IMMEDIATE)

Deletes program lines. The M20 enters Command Mode after a DELETE has been executed.

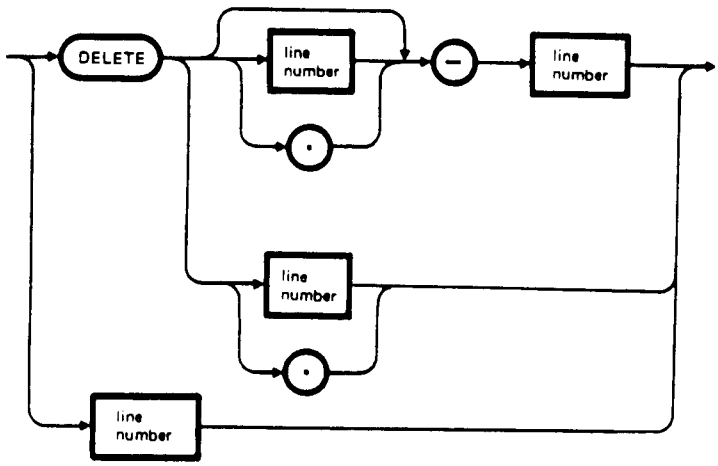


Figure 3-1 DELETE Command

Examples

IF you enter...	THEN...
DELETE . CR	the current line is deleted
5ØØ CR OR DELETE 5ØØ CR	line 5ØØ is deleted
DELETE 1ØØ-2ØØ CR	all lines between 1ØØ and 2ØØ inclusive are deleted
DELETE -4ØØ CR	all lines from the beginning of the program up to and including line 4ØØ are deleted

Note: If any line number specified in a DELETE command is not present in the program, "Illegal function call" will be issued by BASIC.

UPDATING AND MODIFYING A PROGRAM

REPLACING LINES

To change a line you can:

- replace the entire line by entering the number of that line and its new contents

- edit the line using Edit Mode.

First let us use the former method and replace the two mentioned lines of RECTANGLE1 by entering:

```
2Ø  INPUT "Length and Width";L,W CR
5Ø IF W<=Ø THEN 2Ø CR
```

and obtain another listing:

DISPLAY

```
LIST
1Ø REM RECTANGLE1
2Ø INPUT "Length and Width";L,W
3Ø IF L<= Ø THEN 2Ø
5Ø IF W<= Ø THEN 2Ø
6Ø LET AREA=L*W
7Ø PRINT "Area=";AREA;" L=";L;" W=";W
8Ø GOTO 2Ø
9Ø END
Ok
```

COMMENTS

This version of RECTANGLE1 will execute correctly. However to terminate execution you still have to press CTRL C

It is clumsy to have to press CTRL C to terminate execution. We shall, therefore, make some additional modifications.

We can replace statement 8Ø with the following two statements:

1) INPUT "Again:YES=Y,NO=N";X$

2) IF X$="Y" THEN 2Ø

To replace the GOTO statement at line 8Ø, enter:

8Ø INPUT "Again:YES=Y,NO=N";X$ CR

Note: X$ is a string variable.

INSERTING LINES

Now we must insert statement 2) between line 8Ø and 9Ø. We may choose 85 as the line number, entering:

85 IF X$="Y" THEN 2Ø CR

Let us issue another LIST command, and obtain:

DISPLAY	COMMENTS
LIST 1Ø REM RECTANGLE1 2Ø INPUT "Length and Width";L,W 3Ø IF L<=Ø THEN 2Ø 5Ø IF W<=Ø THEN 2Ø 6Ø LET AREA = L*W 7Ø PRINT "Area=";AREA;" L=";L;" W=";W 8Ø INPUT "Again:YES=Y,NO=N";X$ 85 IF X$="Y" THEN 2Ø 9Ø END Ok	This version of RECTANGLE1 does not require that you press CTRL C to stop it. However, the current line numbering is no longer in regular increments of 1Ø

When you run the program this is what happens. After calculating the area of the rectangle whose length and width are entered as input, the program asks if you want to run it again. If you do, you enter Y. When statement 85 is encountered, the program will loop back to statement 2Ø and cycle through the statements again. If you do not want another calculation, you enter N. The condition tested at statement 85 will not be satisfied and the program will continue to the END statement.

RENUMBERING LINES

As we have seen, the current line numbering of RECTANGLE1 is no longer in increments of 1Ø. This is no great drawback for a simple program, but for a complex program for which changes may still be planned, haphazard line numbering can cause problems.

The RENUM command allows you to renumber the lines of a program, starting for example at 1Ø and incrementing each additional line by 1Ø. Simply enter:

R E N U M CR

To see the result, you can issue another LIST command.

```
LIST
1Ø REM RECTANGLE1
2Ø INPUT "Length and Width"; L,W
3Ø IF L<= Ø THEN 2Ø
4Ø IF W<= Ø THEN 2Ø
5Ø LET AREA=L*W
6Ø PRINT "Area=";AREA;" L=";L;" W=";W
7Ø INPUT "Again:YES=Y,NO=N";X$
8Ø IF X$="Y" THEN 2Ø
9Ø END
Ok
```

RENUMBERING AND CROSS-REFERENCES

When a program is resequenced by a RENUM command, all cross-references within the program are updated where necessary. For example, if a program contains the statement GOTO 14Ø and line 14Ø is subsequently renumbered, the reference in the GOTO will be automatically updated to reflect the change.

General Rule

RENUM changes all line number references following GOTO, GOSUB, THEN, ELSE, ON...GOTO, ON...GOSUB and ERL to reflect the new line numbers.

If nonexistent lines are referenced in the program, RENUM causes the following message:

Undefined line xxxxx in yyyyy

The program will be renumbered correctly and the references to nonexistent lines remain unchanged.

RENUM (IMMEDIATE)

Changes the line numbers of the current program.

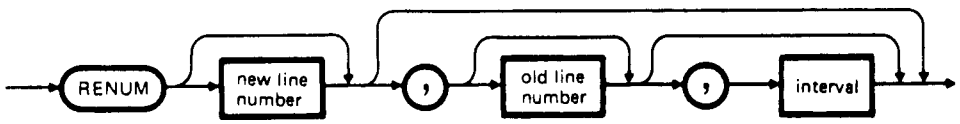


Figure 3-2 RENUM Command

Where

SYNTAX ELEMENT	MEANING	DEFAULT VALUES
new line number	the first new line number	1Ø
old line number	the first old line number	the first line number of the program
interval	the new interval between line numbers	1Ø

Examples

IF you enter...	THEN...
RENUM CR	the entire program is renumbered. The first new line (new line number) is 1Ø and a line interval of 1Ø is assumed (default value)
RENUM 1ØØ CR	the entire program is renumbered. The first new line is 1ØØ and a line interval of 1Ø (default value) is assumed
RENUM 15Ø,,2Ø CR	the entire program is renumbered. The first new line is 15Ø and a line interval of 2Ø is specified

UPDATING AND MODIFYING A PROGRAM

CHANGING LINES WITH THE LINE EDITOR

In Edit Mode it is possible to change portions of a line without re-typing the entire line.

M20 enters Edit Mode if:

- you enter an EDIT command

- a syntax error is detected.

Upon entering Edit Mode, M20 displays the number of the line to be edited, then a space and waits for an Edit Mode command.

The Edit Mode commands do not appear on the screen when you enter them.

In Edit Mode, M20 takes characters as soon as they are entered in - you do not need to press CR.

EDIT (IMMEDIATE)

The EDIT command enters Edit Mode at the specified line.

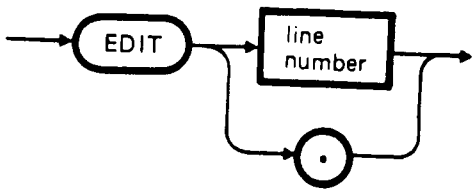


Figure 3-3 EDIT Command

Examples

IF you enter...	THEN M20 displays...
EDIT . CR	nn...n (entering Edit Mode at the current line). Here nn...n means the current line number
EDIT 3ØØ CR	3ØØ (entering Edit Mode at the specified line)

LINE EDIT MODE COMMANDS

The table below summarizes Line Edit Mode commands. They are also grouped in classes.

CLASS	COMMAND	MEANING
to start editing a line	L (List)	causes the current state of the line to be displayed. The current line number is displayed again at the beginning of a new line
	A (Cancel and Start Again)	restores the original line without displaying it. The current line number is displayed again at the beginning of a new line
to move the cursor	SPACE	displays the next character and moves the cursor one position to the right
	CTRL H (Backspace)	erases the last character appearing on the line and moves the cursor one position to the left
to insert characters	I (Insert)	enters Insert State at the current cursor position. You may insert a string of characters. The inserted characters are displayed. To exit Insert State, press CTRL HOME
	X (Extended Line)	causes the remainder of the line to be displayed, moves the cursor to end of line and enters Insert State

	CTRL HOME	exits Insert State but remains in Edit Mode. If you press CR you exit both Insert State and Edit Mode
to delete characters	D (Delete one character)	deletes the next character which is displayed between backslashes (\) and the cursor is positioned to the right
	n D (Delete n characters)	deletes the next n characters. Deleted characters are displayed between backslashes (\) and the cursor is positioned to the right of the last character deleted. If there are fewer than n characters to the right of the cursor, n D deletes the remainder of the line
	H (Hack)	deletes the remainder of the line and enters Insert State
to search characters	S x (Search for the 1st occurrence of x)	searches for the first occurrence of "x" in the line (where "x" is any printable ASCII character) and positions the cursor before it. The character at the current cursor position is not included in the search. If the character is not found in the line, the cursor will stop at the end of

		the line. All characters passed over during the search are displayed
	n **S** **x** (Search for the nth occurrence of x)	is similar to **S** **x** except that it searches for the nth occurrence
	K **x** (Delete until the 1st occurrence of x)	is similar to **S** **x** except that all the characters passed over in the search are deleted. The cursor is positioned before "x" and the deleted characters are enclosed in backslashes (\)
	n **K** **x** (Delete until the nth occurrence of x)	is similar to **K** **x** except that it searches for the nth occurrence
to replace characters	**C** **x** (Change one character)	changes the next character to "x"
	n **C** **x1** **x2**... **xn** (Change n characters)	changes the next n characters to the specified string (keyed after **C**). When you have keyed a string of n characters, Change State is exited and you will return to Edit Mode
to exit Edit Mode	**CR**	causes BASIC to display the new modified line and to return to Command Mode
	E (Exit)	has the same effect as **CR**, but the remainder of the line is not displayed

Q (Quit)	returns to Command Mode and cancels all the changes that were made to the line in the current editing session

Examples

The following table give you some examples for the use of Edit Mode commands.

Note that the cursor is displayed as shown here below (_) when the M20 is in Edit Mode.

STEP	If you enter...	THEN M20 displays...
1	**E** **D** **I** **T** **SPACE** **5** **Ø** **Ø** **CR**	5ØØ _
2	**L**	5ØØ FOR I=1 TO 15 STEP 2 5ØØ _
3	**SPACE** (6 times)	5ØØ FOR I= _
4	**C** **2**	5ØØ FOR I=2_
5	**SPACE** (5 times)	5ØØ FOR I=2 TO 1_
6	**C** **6**	5ØØ FOR I=2 TO 16_
7	**CR**	5ØØ FOR I=2 TO 16 STEP 2 **■**
1	**E** **D** **I** **T** **SPACE** **5** **1** **Ø** **CR**	51Ø _
2	**L**	51Ø LET A(I)=I*SIN(X) 51Ø _
3	**SPACE** (11 times)	51Ø LET A(I)=I*_
4	**3** **C** **C** **O** **S**	51Ø LET A(I)=I*COS_
5	**X** **:** **P** **R** **I** **N** **T** **SPACE** **A** **(** **I** **)** **CR**	51Ø LET A(I)=I*COS(X):PRINT A(I) **■**

1	E D I T SPACE . CR	51Ø _
2	4 D	51Ø \LET _
3	SPACE (11 times)	51Ø \LET \ A(I)=I*COS(_
4	I Y + CTRL HOME	51Ø \LET \ A(I)=I*COS(Y+_
5	SPACE (9 times)	51Ø \LET \ A(I)=I*COS(Y+X):PRINT_
6	4 C I , X ; CR	51Ø \LET \ A(I)=I*COS(Y+X): PRINT I,X; _
7	L I S T SPACE . CR	51Ø A(I)=I*COS(Y+X):PRINT I,X;

EXAMINING CURRENT VARIABLE VALUES

EDITing a program line automatically clears all variable values and closes open data files. If BASIC encounters a syntax error during program execution, it will automatically put you in the Edit Mode. Before editing the line, you may want to examine current variable values. In this case, you must press Q as your first Edit Mode command. This will return you to the Command Mode, where you may examine variable values. Any other Edit Mode command (pressing E, CR etc) will clear out all variables.

RENAMING A FILE

You may change the name of a program or data file residing on a disk with the NAME command, provided there is no write protection. The old filename must exist and the new filename must not exist on the selected volume. After a NAME command is executed, the file exists on the same disk, in the same area of disk space, with the new name. File and volume passwords (if any) are not changed. You must specify the file password and the volume must be enabled (or you must specify the volume password).

NAME (PROGRAM/IMMEDIATE)

Changes the name of a disk file.

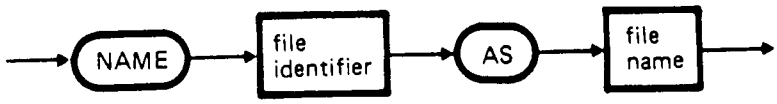


Figure 3-4 NAME Command

Where

SYNTAX ELEMENT	MEANING
file identifier	is either a string constant or a string variable which specifies the program or data file whose name is to be changed
file name	is either a string constant or a string variable which specifies the new name of the file

Examples

Neither the volume nor the file has write protection. The volume is presumed to be enabled.

IF you enter...	THEN...
NAME "1:FR1" AS "FR2" CR	FR1 is changed into FR2. It resides on the diskette inserted in drive 1. FR1 has no password
NAME "VOL1:ACC/PACC" AS "ACC1" CR	ACC is changed into ACC1. It resides on the disk VOL1 which may be inserted in either of the two drives. The file password remains PACC

DELETING A FILE

Program or data files stored on a disk can easily be deleted by use of the KILL command, provided the disk is not write protected. After a file has been deleted, its name can be used again in saving a new file.

You must specify the file password (if any) and the volume must be enabled (or you must specify the volume password).

KILL (PROGRAM/IMMEDIATE)

Deletes a program or a data file stored on a disk.

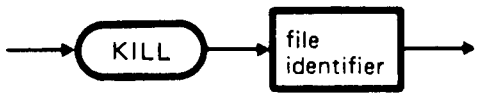


Figure 3-5 KILL Command

Where

File identifier is either a string constant or a string variable which specifies the file to be deleted

Examples

The volume is not write protected and is enabled.

IF you enter...	THEN...
KILL "Business.B" CR	file Business.B is deleted. The search is limited to the last selected drive. The file has no password
KILL "1:Business.B" CR	file Business.B is deleted. The search is limited to the diskette inserted in drive 1. The file has no password

KILL "NUMbers/PNUMØ1" CR — file NUMbers with the password PNUMØ1 is deleted. The search is limited to the last selected drive.

MERGING PROGRAMS

The MERGE command allows you to include a specified program file saved (in ASCII format) on a disk, with the program in memory. MERGE is similar to LOAD, except that the program in memory is not erased before the disk program is loaded. Instead, the disk program is merged into the resident program. That is, program lines in the disk program will simply be inserted into the resident program in sequential order. If a line of the disk program and a line of the resident program have the same line number, the line of the disk program replaces that in memory. The MERGE command must specify the file password (if the disk program has a password) and the volume must be enabled (or you must specify the volume password).

Merging programs may, for instance, be useful to add (standard) subroutines to a program.

It is good programming practice to merge subroutines with line numbers greater than the highest line number of the program. This will improve the MERGE operation speed and allow room to extend the main program.

MERGE (PROGRAM/IMMEDIATE)

Merges the current program with a specified program file (which must have been saved in ASCII format).

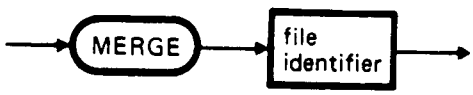


Figure 3-6 MERGE Command

Where

The file identifier is either a string constant or a string variable which specifies an ASCII format program file, i.e. a program saved with the A option.

Examples

DISPLAY	COMMENTS
MERGE "1:Fnew/FnewPASS" CR	the program Fnew with the password FnewPASS is merged with the program in memory Note: The volume resides on drive 1 and is already enabled
MERGE "VØØ1/VPØØ1:FØØ1/PØØ1" CR	the program FØØ1 with the password PØØ1 is merged with the program in memory. Note: The volume VØØ1 is enabled by the use of the password VPØØ1 in the MERGE command

Remark

MERGE closes any open data file and clears variables.

LISTING THE NAMES OF SAVED FILES

If you do not remember the names of program and/or data files residing on a disk, you can use the FILES command to get a listing of them.

The FILES command may be used either with a volume or a file identifier.

When the volume identifier is specified, all the files in the volume are listed (whether they have a password or not).

To execute a FILES command you do not need to know the disk's password, nor does the disk have to be enabled.

When a file identifier is specified, only this file is listed and you need not specify the file password (if any).

Similarly the same functions may be carried out in PCOS with the VQUICK command.

Note: The FILES command does not list passwords.

The information displayed includes:

- the drive on which the disk is currently active
- the name of the disk (if any)
- the amount of file space left on the disk in sectors (a sector is 256 bytes).
- the name of each file on the disk or the name of the specified file, or the name of the selected file(s) if you use the wild card characters "?" or "*" within the file identifier clause. ("?" matches any character, "*" matches any name).

FILES (PROGRAM/IMMEDIATE)

Lists files in the directory of the specified disk.

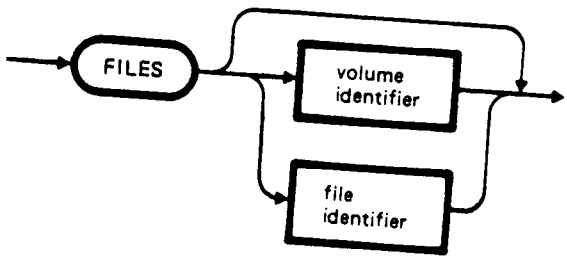


Figure 3-7 FILES Command

Where

SYNTAX ELEMENT	MEANING
volume identifier	is either a string constant or a string variable which specifies the disk whose directory is to be listed
file identifier	is either a string constant or a string variable which specifies the file (in disk directory) to be listed

Examples

IF you enter...	THEN...
FILES CR	the name of each file on the disk (inserted in the last selected drive) is displayed
FILES "Ø:" CR	the name of each file on the diskette inserted in drive Ø is displayed
FILES "1Ø:" CR	the name of each file on the hard disk is displayed
FILES "MYVOL:" CR	the name of each file on the disk MYVOL is displayed. It may be inserted in either of the two drives
FILES "MYVOL/MYPASS:" CR	the name of each file on the disk MYVOL which has the password MYPASS is displayed. It may be inserted in either of the two drives. The specification of the volume password does not affect the execution of this command
FILES "MYFILE" CR	the name of the file MYFILE (which resides on the disk inserted in the last selected drive) is displayed

FILES "1:*.cmd" CR	a list of all the files with the extension '.cmd' residing on the diskette inserted in drive 1 is displayed
FILES "Ø:v???" CR	a list of all the files resident on the diskette inserted in drive Ø with a four letter name beginning with 'v' is displayed.

4. DATA

ABOUT THIS CHAPTER

In this chapter we shall consider how BASIC handles data. We shall look at constants and variables, number representation, numeric conversions and arrays.

CONTENTS

DATA

CONSTANTS AND VARIABLES

Each data item may appear in a BASIC program as either a constant or a variable.

CONSTANTS

Specific numbers such as 15, -2, 3.41 or specific strings such as "AAA.b1", "Cursor***" are referred to as constants. This means that their values remain the same throughout program execution.

VARIABLES

Variables are named data items whose values may change during program execution.

For example, the formula for computing the area of a circle:

3.141592*R∧2

uses variable R. That is R represents any radius and reserves a location in memory for the assignment of a radial value.

Note: The symbol ∧ is an operator which indicates that R is raised to the power specified (2 in this case).

HOW BASIC NAMES VARIABLES

The identifier (or name) of a variable may not be longer than 40 characters. The characters allowed in a variable name are letters and numbers. The period (.) is also allowed. The first character must be a letter. The last character may be a letter, a number, a period, or a type declaration tag (%, !, #, $). The meaning of type declaration tags is illustrated later in this chapter.

Lower case letters in a variable identifier are considered equivalent to their corresponding upper case letters and are converted to their corresponding upper case letters when listing the program.

Examples of variable names are:

STUDENT A1 CCØ1.CLASS ACCOUNT# A$ STRING

Reserved Words

A reserved word (a keyword, a command or a function name), cannot be used as a variable identifier but BASIC permits embedded reserved words within a variable identifier. For example:

```
1Ø PERFORMANCE = 1Ø5.3
2Ø SINGLE = 1371.2
```

are valid program lines, even though PERFORMANCE contains the keyword FOR and SINGLE begins with the name of the built-in function SIN.

REPRESENTATION OF NUMBERS

Numbers are concepts to humans. Most humans are trained to think in base 1Ø. In a computer, numbers are electronic patterns of ones and zeros. The computer performs many of its operations in base 2 (referred to as Binary).

This paragraph gives a review of the concepts of base 2 and of alternative base representation (hexadecimal and octal).

BINARY REPRESENTATION

Before looking at base 2, let us take a look at base 1Ø. The number two hundred and five is represented as:

2Ø5

Base 1Ø uses digits Ø, 1, 2, ...9.The digits have a place value corresponding to powers of ten. The representation above really means:

$$\left(2 \times 1Ø^{2}\right) + \left(Ø \times 1Ø^{1}\right) + \left(5 \times 1Ø^{Ø}\right)$$

The concept of place value also exists in base 2. The difference being that powers of two are represented instead of powers of ten. The number two hundred and five is represented as:

11ØØ11Ø1

Base 2 uses only the digits "1" and "Ø". Therefore, the binary representation shown above means:

$$(1 \times 2^7)+(1 \times 2^6)+(\emptyset \times 2^5)+(\emptyset \times 2^4)+(1 \times 2^3)+(1 \times 2^2)+(\emptyset \times 2^1)+(1 \times 2^\emptyset)$$

This is the same as:

128 + 64 + 8 + 4 + 1 = 2Ø5

A "binary digit" is referred to as a "bit". A bit may be either 1 or Ø.

Bytes

The grouping of 8 bits together is in such common usage that it has been given a special name - a byte. The term byte refers to 8 bits processed as a unit.

The bits of a byte are numbered from Ø (right most, least significant) to 7 (left most, most significant). By doing this, the bit number and the power of two it represents are the same. The following table shows the bit position in a byte and their corresponding values.

BIT POSITION	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT Ø
Meaning	2^7	2^6	2^5	2^4	2^3	2^2	2^1	$2^\emptyset$
Value	128	64	32	16	8	4	2	1

Table 4-1

BASE 1Ø	BASE 2
Ø	Ø
12	11ØØ
27	11Ø11
149	1ØØ1Ø1Ø1
255	11111111

Table 4-2 Conversion Examples

Words

In the M20 data is handled 16 bits (2 bytes) at a time. This quantity is called a "word". The number of bits in a word is machine dependent. The bits in a word are numbered from Ø (right most, least significant) to 15 (left most, most significant).

Another characteristic of a word in the M20 is that two's complement representation is used. Two's complement representation is a method of storing either positive or negative numbers in a word. It works like this:

IF an integer number is...	THEN	AND the word is...
positive	bit 15 is Ø	a positive number represented in normal binary form
negative	bit 15 is 1	a negative number represented in 2's complement form

To find the value of a negative number, you must invert all the bits and add 1 (this will give you its absolute value).

For example:

1 1 1 1 1 Ø Ø 1 1 Ø Ø 1 1 Ø Ø Ø original value (negative)

Inverting all the bits

Ø Ø Ø Ø Ø 1 1 Ø Ø 1 1 Ø Ø 1 1 1 inverted value

Adding 1

Ø Ø Ø Ø Ø 1 1 Ø Ø 1 1 Ø 1 Ø Ø Ø absolute value (inverted + 1)

So the value of the given pattern is:

-164Ø

HEXADECIMAL AND OCTAL REPRESENTATIONS

We have seen that it is possible to represent numbers in decimal (base 1Ø) and binary (base 2). BASIC allows you to represent numbers in octal (base 8) and hexadecimal (base 16) too.

It is often convenient to work with binary numbers but they are tedious to read and write. For this reason we often convert them to octal or hexadecimal.

- base 8, known as "octal" uses one octal digit for three binary digits

- base 16, known as "hex" (short for hexadecimal), uses one hex digit for four binary digits.

The following table shows the decimal (base 1Ø), binary (base 2), octal (base 8) and hex (base 16) representations for the numbers Ø to 16.

DECIMAL	BINARY	OCTAL	DECIMAL	BINARY	HEX
Ø	ØØØ	Ø	Ø	ØØØØ	Ø
1	ØØ1	1	1	ØØØ1	1
2	Ø1Ø	2	2	ØØ1Ø	2
3	Ø11	3	3	ØØ11	3
4	1ØØ	4	4	Ø1ØØ	4
5	1Ø1	5	5	Ø1Ø1	5
6	11Ø	6	6	Ø11Ø	6
7	111	7	7	Ø111	7
8	1ØØØ	1Ø	8	1ØØØ	8
9	1ØØ1	11	9	1ØØ1	9
1Ø	1Ø1Ø	12	1Ø	1Ø1Ø	A
11	1Ø11	13	11	1Ø11	B
12	11ØØ	14	12	11ØØ	C
13	11Ø1	15	13	11Ø1	D
14	111Ø	16	14	111Ø	E
15	1111	17	15	1111	F
16	1ØØØØ	2Ø	16	1ØØØØ	1Ø

Table 4-3

HOW BASIC CLASSIFIES CONSTANTS

The way that BASIC stores a data item determines:

- the amount of memory it will consume

- the speed in which BASIC can process it.

NUMERIC DATA

BASIC can to store all numbers in your program as either:

- Integers (Speed and Efficiency, Limited Range),

- Single precision (General Purpose), or

- Double precision (Maximum Precision, Slowest in Computation).

	INTEGERS	SINGLE PRECISION	DOUBLE PRECISION
Memory Space (bytes)	2	4	8
Range of values	-32768 to 32767	From $\pm 10^{-38}$ To $\pm 10^{38}$	From $\pm 10^{-308}$ To $\pm 10^{308}$
Significant Digits	Up to 5	Up to 7	Up to 16
Displayed Digits (PRINT/LPRINT)	Up to 5	Up to 6 (with rounding)	Up to 15 (with rounding)

Table 4-4

Note: Non significant zeros will not be displayed. For example the value 3.410000 in single precision will be displayed as 3.41.

STRING DATA

Strings (sequences of ASCII characters) are useful for storing non numeric information, such as names, addresses, codes, etc.

For example, the constant:

"FORD ,RENAULT"

is a quoted string constant of 13 characters. Each character in the string (including blank) is stored as an ASCII code, requiring one byte of storage. BASIC would store the above string constant internally as:

ASCII Character	F	O	R	D		,	R	E	N	A	U	L	T
Hex Code	46	4F	52	44	2Ø	2C	52	45	4E	41	55	4C	54

Table 4-5

A string can be up to 255 characters long. A string with length zero is called a "null" string and is represented by a pair of double quotes (""). BASIC allocates strings dynamically, i.e., the memory space reserved for a string may vary during program execution from Ø to 255 bytes.

	NULL STRING	STRING OF n CHAR.	STRING OF MAX. LENGTH
Memory space (bytes)	Ø	n	255
Range of values	—	Any string of printable ASCII characters including blanks	

Table 4-6

NORMAL TYPING CRITERIA TO CLASSIFY CONSTANTS

IF...	THEN...	EXAMPLES
the value is enclosed in double quotes	it is a string	"NO" "YES" "Circle" "" (null string)

the value is not in quotes	it is a number. Note: An exception to this rule is during data input and in DATA statements, where unquoted strings are allowed	521 -15 3.7345E-2 43#
a number is whole and in the range -32768 to 32767	it is an integer constant	1Ø24 721 -32768
the value has the prefix &H and is composed of the numerals Ø-9 and the letters A-F (in the range Ø to FFFF)	it is a hexadecimal constant Note: A hexadecimal constant may be considered an alternative representation of the corresponding integer constant	&H2ØFØ &HF1 &H35 &HFE98 &HFFFF &HØ
the value has the prefix &0 or & and is composed of the numerals Ø-7 (in the range Ø to 177777)	it is an octal constant Note: An octal constant may be considered an alternative representation of the corresponding integer constant	&07Ø &044 &71175
a number is not an integer and contains 7 or fewer digits	it is single precision	-2.3 32768 45.314 -65ØØØ
a number contains more than 7 digits	it is double precision	52174593 -54.397124 8.799999999

TYPE DECLARATION TAGS

You can override BASIC's normal typing criteria by adding the following "tags" to the end of a numeric constant.

TAG	MEANING	EXAMPLES
!	makes the number single precision	5.7211Ø333! the constant is classified as single precision and shortened to 7 digits (i.e., 5.7211Ø3)
E	single precision floating point. The E indicates the constant is to be multiplied by a specified power of 1Ø	7.31E4 means $7.31 \times 1Ø^{4}$ i.e. 731ØØ
#	makes the number double precision	4# 5.21#
D	double precision floating point. The D indicates the constant is to be multiplied by a specified power of 1Ø	7.2D-3 means $7.2 \times 1Ø^{-3}$ i.e. Ø.ØØ72

HOW BASIC CLASSIFIES VARIABLES

When BASIC encounters a variable identifier in a program, it classifies it as either a string, integer, single or double precision number.

BASIC classifies all variable names as single precision initially. For example, if this is the first line of your program:

```
1Ø X1 = 3.5
```

BASIC classifies X1 as a single precision variable.

However, you may assign different type attributes to variables using either definition statements (DEFtype statements) or a type declaration tag at the end of the variable identifier.

DEFINT/DEFSNG/DEFDBL/DEFSTR (PROGRAM/IMMEDIATE)

Four DEFtype statements are provided to assign different types to variables.

A DEFtype statement declares that the variable names beginning with the letter(s) specified will be that type variable.

DEFtype statements are usually placed at the beginning of your program, and must precede the use of the defined variables.

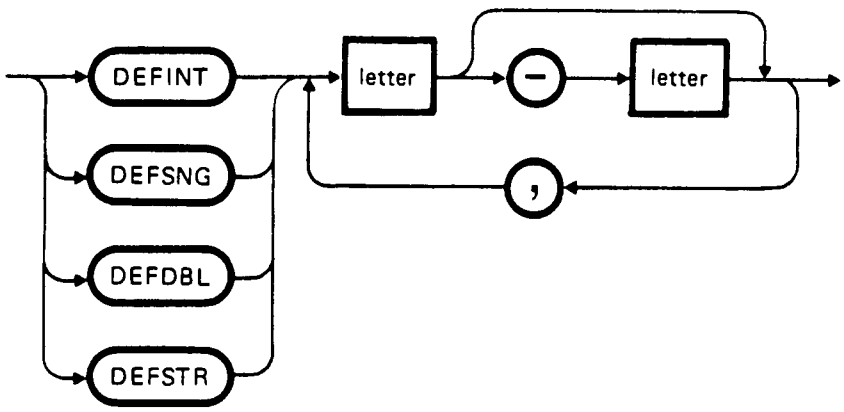


Figure 4-1 DEFtype Statements

Default Values

Unless otherwise specified all program variables are assumed to be single precision.

Examples

IF you enter...	THEN...
1Ø DEFINT A-Z CR	all program variables will be integer
1Ø DEFDBL D CR	all program variables beginning with the letter D will be double precision

```
10 DEFSTR S,U-W CR
```

all program variables beginning with the letters S, U, V and W will be string variables

TYPE DECLARATION TAGS

As with constants, you can always override the type of a variable name by adding a type declaration tag at the end. There are four type declaration tags for variables:

TAG	MEANING	EXAMPLES
%	integer	A% STEP% INCREMENT% are all integer variables, regardless of what attributes have been assigned to the letters A, S and I
!	single precision	SPEED! SPACE! TIME! are all single precision variables, regardless of what attributes have been assigned to the letters S and T
#	double precision	TOTAL# SUBTOTAL# X1# are all double precision variables, regardless of which attributes have been assigned to the letters T, S and X

$	string	A$ B1$ NAME.CLASS$ are all string variables, regardless of which attributes have been assigned to the letters A, B and N

NUMERIC CONVERSIONS

Often a program or immediate line might ask BASIC to assign one type of constant to a different type of variable. For example, if you enter:

I%=5.31 CR

BASIC will first round the single precision constant 5.31 to the nearest integer to assign it to the integer variable I%. Thus the value of I% will be 5.

You may also want to convert one type of variable to a different type of variable. such as:

SCALE!=B% CR
SECONDS!=C1# CR
BOX#=W% CR

The conversion procedures are illustrated in the examples on the following pages.

SINGLE OR DOUBLE PRECISION TO INTEGER

BASIC converts the original value to an integer by rounding the fractional part.

Note: The rounded value must be greater than or equal to -32768 and less than 32767, otherwise an Overflow error occurs.

Examples

DISPLAY	COMMENTS
C%=-15.1 Ok ?C% -15 Ok	-15 is assigned to C%
C%=4.1E2 Ok ?C% 410 Ok	410 is assigned to C%
C%=47.8 Ok ?C% 48 Ok	48 is assigned to C%
C%=7.21473D-3 Ok ?C% 0 Ok	0 is assigned to C%
C%=-32768.5 Overflow Ok	an Overflow error occurs

INTEGER TO SINGLE OR DOUBLE PRECISION

No error is introduced. The converted value looks like the original value with zeros to the right of the decimal point.

DISPLAY	COMMENTS
S!=326 Ok ?S! 326 Ok	326 is stored in S! as 326.0000 but it is displayed as 326
D#= 326 Ok ?D# 326 Ok	326 is stored in D# as 326.0000000000000 but it is displayed as 326

SINGLE TO DOUBLE PRECISION

BASIC adds trailing zeros to the single precision number.

If the original value:

- has an exact binary representation, no error will be introduced

- does not have an exact binary representation, an arithmetic error is introduced when converting the value.

Examples

DISPLAY	COMMENTS
B#=1.5 Ok ?B# 1.5 Ok	when entering B#=1.5, you store 1.50000000000000000, in B# but 1.5 is displayed. Note: 1.5 has an exact binary representation
C#=1.3 Ok ?C# 1.29999995231628 Ok	When entering C#=1.3 you store 1.299999952316280 in C# but it is displayed as 1.29999995231628. Note: 1.3 does not have an exact binary representation

Remarks

To avoid losing accuracy you should keep single to double precision conversions out of your programs. For example, whenever you assign a constant value to a double-precision variable, you can force the constant to be double-precision.

B#= 1.3# B#= 1.3D

Both store 1.3 in B#.

When the single-precision value is stored in a variable, convert the single-precision variable to a string with STR$ function (see Chapter 9), then convert the resultant string back into a number with VAL (see Chapter 9).

DISPLAY

```
LIST
10 B!=1.3
20 B#=B!
30 PRINT B#
Ok
RUN
 1.29999995231628
Ok
```

COMMENTS

This program displays the value of B# losing accuracy.

```
LIST
10 B!=1.3
20 B#=VAL(STR$(B!))
30 PRINT B#
Ok
RUN
 1.3
```

This program displays the value of B# without losing accuracy.

DOUBLE TO SINGLE PRECISION

This involves converting a number with up to 16 significant digits into a number with no more than 7.

Only the first seven digits, rounded of the converted value, will be valid.

Before displaying or printing such a number BASIC rounds it to six digits.

Note: If the double precision value is outside the range of single precision values an Overflow error occurs.

Example

DISPLAY	COMMENTS
P!=2.Ø3999996 Ok ?P! 2.Ø4 Ok	2.Ø4ØØØØ is stored in P! but is is displayed as 2.Ø4

ILLEGAL CONVERSIONS

You cannot convert numeric values to string or vice versa by an assignment statement. For example:

C$=321.7

is illegal. (Use STR$ and VAL functions to accomplish such conversions. See Chapter 9).

SUBSCRIPTED VARIABLES AND ARRAYS

As mentioned before (see Chapter 1) a variable may be a simple variable or a subscripted variable. Subscripted variables are elements of an "array".

An array is a collection of variables of the same type under one name. You can distinguish them by the value(s) of one or more subscripts appearing in parentheses after the array name. For example, if A is a one dimensional array, A(Ø) is the first element, A(1) the second element, and so on (supposing that the subscript lower bound is Ø).

A subscript value must be a positive integer number, but any numeric expression whose value is positive may be entered as a subscript. If its value is not an integer, it is rounded to an integer.

DATA

An array may have any number of dimensions. A one dimensional array might be thought of as a list of items. There may be many rows but only one column. A two dimensional array is like a table of values. There may be several rows and several columns of items.

To define an array you must:

- give it a name (any valid variable name may be assumed)

- establish the upper and lower subscript bounds.

To do that you have to use a DIM statement, and optionally an OPTION BASE statement.

If you specify in a program:

1Ø OPTION BASE 1

The lower bound of all arrays is 1.

If you omit the OPTION BASE statement, or if you specify OPTION BASE Ø, the lower bound of all arrays is Ø (the default lower bound).

It is also possible to re-define an array, by writing an ERASE statement before a DIM statement (see below).

ONE DIMENSIONAL ARRAYS

Suppose we have the following list of numbers:

17, -9, 32, 1Ø5, -48

If you define a one dimensional numeric array V, you can store all the values in the list introducing only one array variable and you can access each array element by specifying the appropriate subscript.

Array V

Element	Contents
V(Ø)	17
V(1)	-9
V(2)	32
V(3)	1Ø5
V(4)	-48

Each element in Array V is specified by its subscript. For example V(1) is -9 and V(3) is 1Ø5. The subscript identifies the location of the element in the array.

MULTI DIMENSIONAL ARRAYS

We may use a two dimensional array to store the values of a table. Suppose we have the following table:

NAME	CODE	COUNTRY	SEX
Anna	21SAA	Great Britain	F
John	35ECK	USA	M
Richard	7ØWST	Sweden	M

Table 4-7

This table contains 3 rows and 4 columns for a total of 12 string values.

If you define a string array A$ you can store all the values in the table introducing only one array variable and you can access any array element by specifying the appropriate subscripts.

SUBSCRIPT	Ø	1	2	3
Ø	Anna	21SAA	Great Britain	F
1	John	35ECR	USA	M
2	Richard	7ØWST	Sweden	M

Table 4-8 Array A$

Each element in array A$ is specified by its location in the array with two subscripts, separated by a comma and enclosed within parentheses. The first subscript designates the "row" in the array; the second subscript designates the "column". For example:

A$(Ø,1) is the string 21SAA
A$(2,3) is the character M

You may define arrays with even more dimensions, but they are rarely used.

DIM (PROGRAM/IMMEDIATE)

Specifies the array name, the number of dimensions and the subscript upper bound per dimension. The DIM statement may specify one or more arrays.

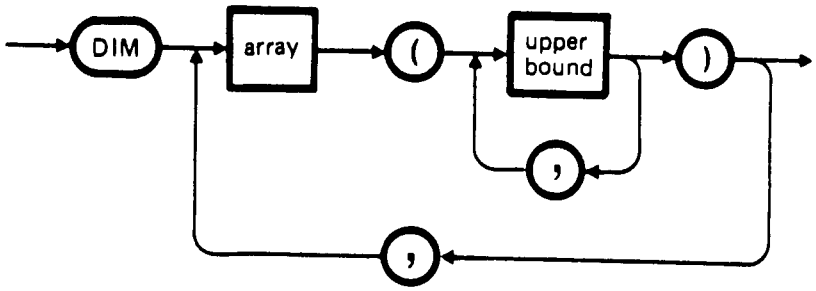


Figure 4-2 DIM Statement

Where

SYNTAX ELEMENT	MEANING	DEFAULT VALUES
array	is the array name. Any legal variable name may be used	

upper bound	is any positive numeric constant or variable. If it is not an integer, it is rounded to the nearest integer	if no DIM is specifed, an upper bound of 1Ø is assumed for each dimension and the number of dimensions are set when you refer to an array element in your program

Example

IF you enter...	THEN...
1Ø DIM A(5), B$(2Ø,3Ø) CR	you set up a one dimensional array A with subscripts from Ø to 5, and a two dimensional string array B$ with subscripts from Ø,Ø to 2Ø, 3Ø. Note: A is numeric, unless differently stated by a DEFSTR statement

Number of Dimensions

With BASIC, you may have as many dimensions in your array as you like, depending on the available memory. One and two dimensional arrays are the most frequently used.

If no DIM is specified, the first reference to an array element in the program will create the array with the specified number of dimensions. For example, if a program statement refers to:

AR1(3,5,1Ø)

Then AR1 is created with 3 dimensions and a default upper bound of 1Ø for each dimension.

DATA

Number of Elements per Dimension

IF...	AND IF...	THEN...
no DIM is used	OPTION BASE Ø is set	11 elements (subscripts Ø-1Ø are allowed in each dimension)
	OPTION BASE 1 is set	1Ø elements (subscripts 1-1Ø are allowed in each dimension)
DIM is used	OPTION BASE Ø is set	the number of elements in each dimension is calculated by adding 1 to each upper bound subscript
	OPTION BASE 1 is set	the number of elements in each dimension coincides with each upper bound subscript

To Define an Array

YOU MUST...	AND EITHER...	OR...
establish the subscript lower bound	use an OPTION BASE 1 statement	adopt the default OPTION BASE Ø
assign a name to the array	use a DIM statement	refer an array element within the program.
establish the number of dimensions		Note: In this case a subscript upper bound of 1Ø for each dimension is assumed.
establish the subscript upper bound per dimension		

Remarks

- the DIM statement sets all the elements of the specified arrays to an initial value of zero

- a DIM statement cannot be preceded by an array reference

- a DIM statement does not set the subscript upper bound per dimension, in case it is jumped over. For example:

DISPLAY	COMMENTS
LIST 1Ø I=1 2Ø GOTO 4Ø 3Ø DIM A(5Ø) 4Ø A(1Ø)=3 5Ø A(11)=45 Ok RUN Subscript out of range in 5Ø Ok	The M20 will display: Subscript out of range in 5Ø when statement 5Ø is executed, as statement 3Ø is jumped over and an upper bound of 1Ø is assumed by default

ERASE (PROGRAM/IMMEDIATE)

Releases space and variable names previously reserved for arrays. The data is lost and the array(s) no longer exist.

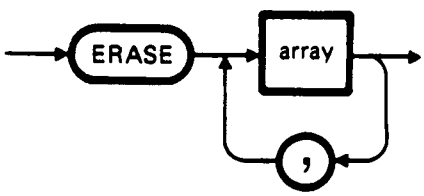


Figure 4-3 ERASE Statement

Example

DISPLAY	COMMENTS
1Ø DIM A(15,15),B(1Ø,2Ø) : 1ØØ ERASE A,B 11Ø DIM A (1ØØ),B(2,2,2)	upon execution of statement 1ØØ, arrays A and B are deleted and the corresponding memory space is made free. You may define other arrays (see statement 11Ø) with the same names but different numbers of dimensions and upper bounds

Remarks

It is not normally good programming practice to reuse an identifier. This may generate errors or reduce the program readability. You may, however, find it useful to redeclare an erased array; for example, when an array name is known by a subroutine and you want to pass arrays with different number of dimensions or subscript upper bounds to this subroutine.

OPTION BASE (PROGRAM/IMMEDIATE)

Declares the lower bound for array subscripts.

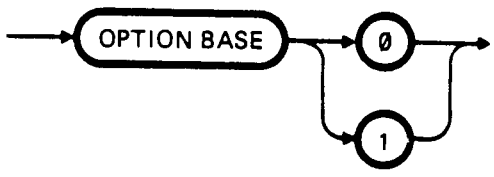


Figure 4-4 OPTION BASE Statement

Default Value

OPTION BASE Ø is assumed by default (i.e. if you do not write any OPTION BASE statement in your program.)

Example

IF you enter...	THEN...
1Ø OPTION BASE 1 CR OR OPTION BASE 1 CR (in immediate mode)	the lower bound of all arrays is 1

Remarks

You will find the OPTION BASE 1 useful when converting programs from other machines to your M20. Many older BASICs number all arrays from 1.

The OPTION BASE statement cannot be preceded by a DIM statement or by an array reference.

5. HOW BASIC INPUTS DATA

ABOUT THIS CHAPTER

This chapter will describe some ways to supply data to the computer via your program.

We shall examine:

- the CLEAR, LET and SWAP statements

- the INPUT and LINE INPUT statement

- the DATA, READ and RESTORE statements.

Other ways to supply data, using external files, will be examined later, (see Chapter 12).

CONTENTS

HOW BASIC INPUTS DATA

ASSIGNMENT STATEMENTS

There are three assignment statements in BASIC:

- the CLEAR statement, which allows you to set all numeric variables to zero and all string variables to null.

- the LET statement, which allows you to assign the value of an expression to a variable. The variable and the expression must be either both numeric or both string

- the SWAP statement, which allows you to exchange the values of two variables, provided they are the same type (integer, single-precision, double-precision, string).

LET and SWAP are often used as immediate statements for quick computations.

CLEAR (PROGRAM/IMMEDIATE)

Sets all numeric variables to zero, all string variables to null, closes all open data files and windows (see Chapter 14) and clears the screen.

CLEAR optionally sets the amount of user memory available for BASIC programs and the amount of stack space.

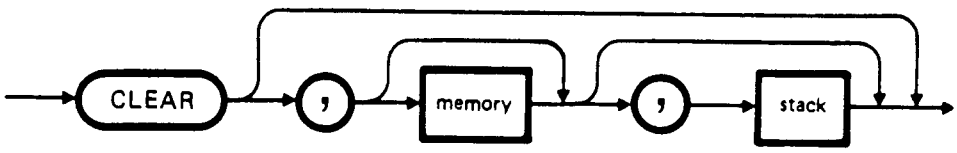


Figure 5-1 CLEAR Statement

Where

SYNTAX ELEMENT	MEANING
memory	sets the amount of memory available for BASIC programs. This value may also be set by the PCOS

	command SBASIC. If omitted, its value is either that established by the SBASIC command, or 37ØØØ (as a second alternative).
stack	sets aside stack space for BASIC. The default value is 512 bytes or one-eighth of the available memory whichever is smaller. The stack is a part of memory available for BASIC used to store return addresses of subprograms, functions etc.

Examples

DISPLAY	COMMENTS
CLEAR	clears variables, closes data files and windows, and clears the screen. The memory is either that established by the SBASIC command, or 37ØØØ bytes. The stack is assumed by default.
CLEAR ,32768	as in the example above, but memory is set to 32768 bytes.
CLEAR ,,2ØØØ	as in the first example, but stack is set to 2ØØØ bytes.
CLEAR ,32768,2ØØØ	as in the first example, but memory is set to 32768 bytes and stack to 2ØØØ bytes.

Remarks

BASIC automatically sets all numeric variables to zero and all string variables to null at the beginning of the execution of a program (except variables defined in the COMMON area, if the program is CHAINed to another, see Chapter 11).

BASIC allocates string space dynamically. An "Out of string space" error occurs only if there is no free memory left for BASIC to use.

LET (PROGRAM/IMMEDIATE)

Assigns a value to a variable.

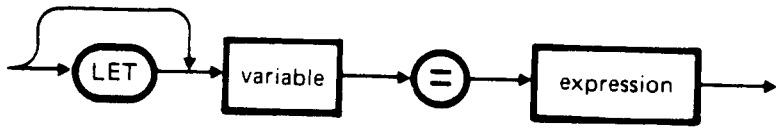


Figure 5-2 LET Statement

Examples

IF you enter ...	THEN...
LET K = 1.5 CR	the value 1.5 is assigned to the numeric variable K
LET X = K + 2 CR	the value of the numeric expression K + 2 is assigned to the numeric variable X
A$(I) = "ABC" CR	the value of the string constant "ABC" is assigned to the subscripted string variable A$(I).

Note: The keyword LET is optional

Numeric Assignments

If the data-type of the value resulting from the evaluation of the numeric expression is different from the type of the receiving variable, BASIC converts the type of the expression value to the type of the receiving variable, following the rules we have just seen (see NUMERIC CONVERSIONS paragraph in Chapter 4).

Rounding or overflow may occur, if the receiving variable is not able to contain the computed value.

String Assignment

String assignment is performed by moving the string expression value character by character into the receiving variable. The operation ends when all the characters have been moved.

Remarks

Simultaneous assignments are not allowed. If you enter for instance:

```
10Ø LET B% = C% = Ø CR
```

BASIC WOULD INTERPRET THE SECOND EQUAL SIGN AS A RELATIONAL OPERATOR and set B% equal to -1 (i.e. true) if C% equalled Ø, and Ø (i.e. false) if C% is different from zero (for a fuller explanation of relational expressions see Chapter 6).

SWAP (PROGRAM/IMMEDIATE)

Allows you to exchange the values of two simple variables. Any type of variable may be SWAPped (integer, single-precision, double-precision, string) but the two variables must be of the same type or a "Type mismatch" error occurs. They must also be initialized, or an "Illegal function call" error occurs.

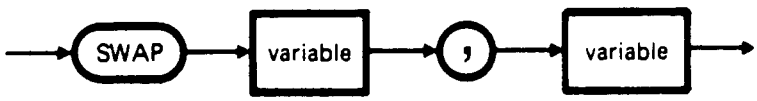


Figure 5-3 SWAP Statement

Example

DISPLAY	COMMENTS
LIST 1Ø A$ = " ONE " 2Ø B$ = " ALL " 3Ø C$ = " FOR "	Statement 5Ø SWAPs the values of A$ and B$, statement 4Ø displays ONE FOR ALL, statement 6Ø displays ALL FOR ONE.

HOW BASIC INPUTS DATA

```
4Ø PRINT A$;C$;B$
5Ø SWAP A$,B$
6Ø PRINT A$;C$;B$
Ok
RUN
ONE FOR ALL
ALL FOR ONE
Ok
```

THE INTERNAL DATA FILE

Many problems require that a large number of constants be entered into the computer. To do this, you could use many LET, INPUT, or LINE INPUT statements.

It is clear, though, that this would be arduous, if you had a long list of data to be entered. A much more convenient and effective way to enter such constants is by using the DATA, READ, and RESTORE statements. DATA statements create an "internal" file, i.e. a sequence of data belongs to the program, which must be transferred into the program variables by one or more READ statements. The RESTORE statement allows you to reposition the pointer at the beginning of the file or to a specified line number.

DATA/READ/RESTORE (PROGRAM)

DATA creates an internal data file.

READ reads data from one or more DATA statements into the specified variables.

RESTORE moves the pointer either to the beginning of an internal data file (created by one or more DATA statements) or to a specified line number.

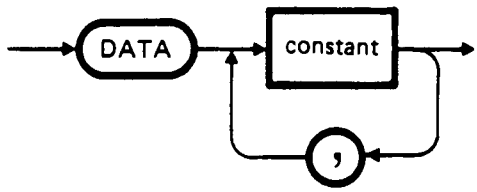


Figure 5-4 DATA Statement

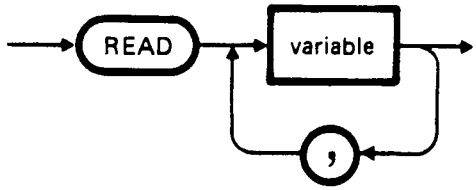


Figure 5-5 READ Statement

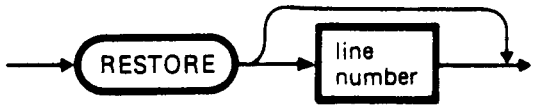


Figure 5-6 RESTORE Statement

Examples

DISPLAY	COMMENTS
LIST	the values 1 to 1Ø are assigned to ten variables
1Ø READ A,B,C,D,E,F,G,H,I,J	
2Ø DATA 1,2,3,4,5,6,7,8,9,1Ø	
3Ø PRINT A;B;C;D;E;F;G;H;I;J	
Ok	
RUN	
1 2 3 4 5 6 7 8 9 1Ø	
Ok	

```
LIST
1Ø DATA 1,2,3,4
2Ø READ A,B,C,D,E,F,G,H,I,J
3Ø DATA 5,6,7
4Ø DATA 8,9,10
5Ø PRINT A;B;C;D;E;F;G;H;I;J
Ok
RUN
 1  2  3  4  5  6  7  8  9  1Ø
Ok
```

statements 1Ø, 2Ø, 3Ø, and 4Ø have the same effect as statements 1Ø and 2Ø in the previous example.

Note: A DATA statement in a program need not correspond to a specific READ statement. This is because before program execution, a data file (the "internal file" as it is often called) is created. It contains all the values of all the DATA statements in the program in line number sequence. When the program is executed, READ takes its values from this file

```
LIST
1Ø READ A,B,C
2Ø DATA 1,2,3,4,5,6,7,8,9,1Ø
3Ø PRINT A;B;C
4Ø READ D,E,F,G
5Ø PRINT D;E;F;G
RUN
 1  2  3
 4  5  6  7
Ok
```

statement 1Ø assigns the values 1,2, and 3 to A,B,and C; statement 4Ø assigns the values 4,5,6 and 7 to D,E,F and G respectively.

Note: When you access a data file you do not have to read all the values stored in it

```
LIST
1Ø READ A.B,C,D,E
2Ø DATA 1,2,3,4
Ok
RUN
Out of data
Ok
```

M20 displays an error message:

Out of data

and returns to Command Mode, because there are fewer data items than variables

```
LIST
1Ø READ A,B,C
2Ø DATA 15,25,35,5,6,12
3Ø PRINT A;B;C
4Ø RESTORE
5Ø READ X,Y,Z
6Ø PRINT X;Y;Z
Ok
RUN
 15  25  35
```

statement 1Ø causes the variable A to be assigned the value 15, B the value 25, and C the value 35. The RESTORE statement at line 4Ø will cause values to be assigned starting from the beginning of the file again. Hence, statement 6Ø causes the very same values assigned to A,B, and C, (15,25,35) to be assigned, respectively, to X, Y,

```
 15  25  35
Ok
```

and Z. If RESTORE were not present, X would be assigned the value 5, Y the value 6, and Z the value 12

```
LIST
1Ø READ X1$, Y1$, Z1
2Ø DATA "DENVER,", COLORADO, 8Ø211
3Ø PRINT X1$;Y1$;Z1
Ok
RUN
DENVER,COLORADO 8Ø211
Ok
```

statement 1Ø causes X1$ to be assigned the value DENVER, (including the final comma), Y1$ the value COLORADO, and Z1 the value 8Ø211.

Note: READ statements may contain both numeric and string variables, DATA statements may contain both numeric and string data.

The data-type of an entry in the data sequence must correspond to the type of the variable to which it is to be assigned; i.e., numeric variables require numeric constants as data (conversion from one numeric type to another is allowed, for example you may have a single precision floating point constant associated with an integer variable) and string variables require quoted or unquoted strings as data. A quoted string is required if the string contains commas (e.g. DENVER,) or initial or final blanks (e.g. the blank preceeding COLORADO in statement 2Ø is skipped as COLORADO is not a quoted string)

INPUT STATEMENTS

The DATA statement uses constants to assign values to variables. You must know, when you are entering your program, what values you want to assign. Furthermore the values contained in the internal data file are saved whenever your program is saved. Hence, these values are relatively permanent; they can be changed only by changing one or more DATA statements in the program.

HOW BASIC INPUTS DATA

The INPUT and a LINE INPUT statements, offer you more flexibility. Using them you enter values only when the program is executed. When one of these statements is encountered, program execution is suspended and M20 waits for you to enter data from the keyboard.

As a consequence, after you have saved a program, you can run it at any time, and supply values to the computer on the spot, without changing the program itself. This flexibility allows you to write a general program to solve a particular problem before you know the specific values the program will use. However, if you have a lot of data to enter, it is better to use an internal file (permanent data) on one or more external files (see Chapter 12).

The INPUT statement allows you to enter one or more numeric or string data-items (separated by a comma). They will be assigned to the variable(s) specified in the statement. The LINE INPUT statement allows you to enter an entire input line and assign it to a string variable.

You may insert a prompt message in both INPUT and LINE INPUT statements. This will be displayed on the screen when the statement is executed to remind you what to enter.

INPUT (PROGRAM)

Reads data-item(s) from the keyboard and assigns it/them to one or more specified variables.

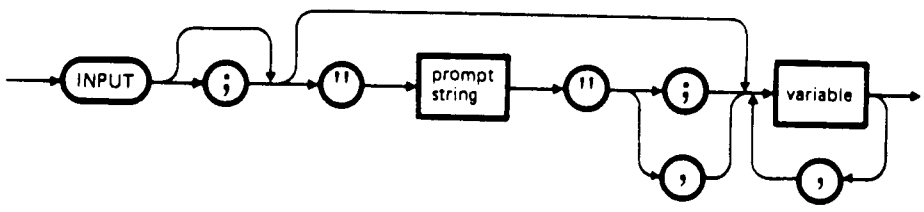


Figure 5-7 INPUT Statement

A Question Mark

A question mark (followed by a blank) is automatically displayed as a standard prompt when executing an INPUT statement, even though the statement does not include a prompt-string.

DISPLAY

```
LIST
1Ø INPUT X
2Ø PRINT X "SQUARED IS" X^2
3Ø END
Ok

RUN
? 5
 5 SQUARED IS 25
Ok
```

COMMENTS

When executing statement 1Ø the standard prompt (?) is displayed, indicating that the program is waiting for data.

No prompt string is used in the INPUT statement in this case (see statement 1Ø)

Self Prompting

By inserting a prompt-string in an INPUT statement, you may prompt for each value required.

DISPLAY

```
LIST
1Ø PI = 3.1415
2Ø INPUT "Radius";R
3Ø A=PI*R^2
4Ø PRINT "Area";A
5Ø GOTO 2Ø
Ok
RUN
Radius? 7.4
Area 172.Ø29
Radius?
etc.
```

COMMENTS

the user prompt (Radius) is displayed before the standard prompt (?), when statement 2Ø is encountered

To Suppress the Standard Prompt

You may suppress the standard prompt (?) by writing a comma (,) after your prompt.

HOW BASIC INPUTS DATA

DISPLAY	COMMENTS
LIST 1Ø INPUT "Date ", D$ 2Ø PRINT D$ Ok RUN Date 3Ø/Oct/69 3Ø/Oct/69 Ok	the standard prompt (?) is suppressed because a comma (,) - instead of a semicolon (;) - follows the user prompt in statement 1Ø

To Suppress the Echo of CR

You may suppress the echo of CR on the screen, by writing a semicolon (;) after INPUT.

DISPLAY	COMMENTS
LIST 1Ø INPUT; "Date";D$ 2Ø PRINT " J.C." Ok RUN Date? 3Ø/Oct/69 J.C.	the echo on the screen of the carriage-return/ line-feed is suppressed by inserting a semicolon (;) immediately after INPUT (see statement 1Ø) The next PRINT/INPUT operation will be executed from the next screen position (see statement 2Ø)

To Enter a List of Data

An INPUT statement allows you to enter one or more numeric or string data from the keyboard.

DISPLAY	COMMENTS
LIST 1Ø INPUT A,B$,C(3) 2Ø PRINT A;B$;C(3) 3Ø GOTO 1Ø Ok RUN ? 1.2,ABC,4 1.2 ABC 4	when statement 1Ø is executed, you must enter three data-items. The first must be numeric (1.2), the second string (ABC) (and need not to be surrounded by quotation marks), the third (4) numeric. They will be assigned to variables A, B$ and C(3) respectively.

```
? ABD,1.3,5
?Redo from start
? 1.3,ABD,5
 1.3 ABD 5
?^C
Break in 10
Ok
```

When statement 10 is executed for the second time, suppose that you enter a datum of the wrong type, (ABD) i.e. a string instead of a number. The system displays:

? Redo from start

and you must re-enter the value.

To interrupt program execution press CTRL C
To resume execution press C O N T CR.

Note: The data-type of a keyboard entry must correspond to the type of the variable to which it is to be assigned; i.e. numeric variables require numeric constants as data (conversion from one numeric type to another is allowed, for example you may enter a double precision floating point constant to initialize an integer variable) and string variables require quoted or unquoted strings as data. A quoted string is required if the string contains commas or initial or final blanks. Numeric items may be input into string variables. If you input a number into a string and then you wish to re-obtain its numeric value use the VAL function (see Chapter 9), to prevent type mismatch errors

?Redo from Start

Responding to INPUT with too many or too few items, or with the wrong type of value (string instead of numeric) causes the message "?Redo from start" to be displayed . No assignment of input values is made until an acceptable response is given.

LINE INPUT (PROGRAM)

Inputs an entire line up to a carriage return/line feed and assigns it to a string variable, without the use of delimiters (255 characters is the maximum length of a line).

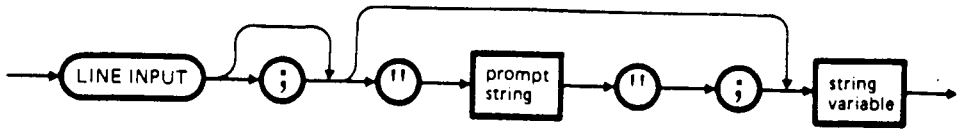


Figure 5-8 LINE INPUT Statement

A Question Mark within the Prompt

The standard prompt (?) does not appear when executing a LINE INPUT statement. You can close your own prompt with a question mark if you wish.

DISPLAY	COMMENTS
LIST 10 LINE INPUT "Name? ";N$ 20 PRINT "JONES" Ok RUN Name? LINDA JONES Ok	the prompt string (Name?) is displayed before input is accepted. All input from the end of the prompt to CR is assigned to the string-variable (N$)

To Suppress the Echo of CR

You may suppress the echo of CR on the screen, by writing a semicolon (;) after LINE INPUT.

DISPLAY	COMMENTS
LIST 10 LINE INPUT;"Name? ";N$ 20 PRINT " JONES" Ok RUN Name? LINDA JONES Ok	CR does not echo a carriage return/line-feed, as LINE INPUT (see statement 10) is followed by a semicolon (;). The next PRINT/INPUT operation (see statement 20) will be executed from the next screen position

6. EXPRESSIONS

ABOUT THIS CHAPTER

This chapter classifies BASIC expressions as numeric, string, relational or logical. It gives the rules the user must respect in forming expressions as well as the priority rules BASIC assumes in evaluating them.

CONTENTS

EXPRESSIONS

NUMERIC EXPRESSIONS

Most of the programs you write will involve some numeric calculations.

As you may have noticed in our examples, only variables appear to the left of the equal sign in LET statements.

Both variables and constants, however, can appear to the right of the equal sign. They can, in fact, be connected by means of special symbols, called operators, to indicate numeric operations. Some examples follow:

```
 7Ø LET L=ACCOUNT
 6Ø LET Y = 16+1.7+12
2ØØ M = 83-44+37/N
 2Ø LET X = X+1
```

The last statement is particularly interesting. Most LET statements look like algebraic equations; this last one does not. The equation X = X+1 makes no sense algebraically. The LET statement assigns a value to a variable: it does not imply that the values to the left and right of the equal sign are mathematically equal. This last statement, which is valid and meaningful, can be interpreted as follows: add 1 to the value represented by variable X, and assign this new value to X. This new value of X will replace the old one. Thus the equal sign is itself an operator.

That part of the LET statement to the right of the equal sign is called an expression. The expression specifies the value to be assigned to the variable to its left. (The evaluation of an expression yields a single numeric value.) A numeric expression can be composed of a single number or a single numeric variable as well as some combination of numbers, numeric variables, and operators. Remember, however, that a numeric variable must be assigned a value before it is used in an expression. If it is not, the variable automatically assumes the value Ø.

Some examples of numeric expressions are shown below:

```
X
X+Y+SPEED
6.4 ^ 2
-7+A/CYAR
```

Numeric Operators

As mentioned before, BASIC uses operators to indicate numeric operations. There are eight numeric operations, each with its own symbol.

SYMBOL	OPERATION	EXAMPLES
+	addition	X = 3.2 Ok ?X+1.1 4.3 Ok
-	subtraction	?X-1.3 1.9 Ok
\	integer division. The operands are rounded to the nearest integers (which must be in the range - 32768 to 32767) before the division is performed, and the quotient is truncated to an integer	?10\4 2 Ok ? 25.68\6.99 3 Ok
MOD	modulus arithmetic. It gives the integer value which is the remainder of an integer division	?1Ø.4 MOD 4 2 Ok (1Ø/4 = 2 with remainder 2) ?25.68 MOD 6.99 5 Ok (26/7 = 3 with remainder 5)
*	multiplication	?X*3.92 12.544 Ok
/	Division	?3/6.05 Ø.495868 Ok
-	negation It changes the sign of the operand	?-X -3.2 Ok

EXPRESSIONS

^	Exponentiation	?X ^ 3 32.768 Ok

Remarks

Be sure to include the * when specifying multiplication. In mathematics, 6X is valid; in BASIC, 6*X must be written to express 6 times X.

For your convenience, all the numeric operators used in BASIC have been placed on the M20 Keyboard both in the numeric and alphanumeric section (except the exponentiation symbol, which appears in the alphanumeric section only and MOD which must be entered typing its three characters).

Numeric Operator Priority

When two or more operators are used in an expression, it often seems ambiguous. For example, does the expression:

3*L - 6*W

mean

(3*L) - (6*W)

or

3* (L - 6*W)?

BASIC has built-in priorities for performing different numeric operations.

Numeric operations and priority rules are as follows (in order of descending priority).

PRIORITY	OPERATION	COMMENTS
HIGHEST	exponentiation	
	negation	

	multiplication and Division	Multiplication and division have the same priority
	integer division	
	modulus arithmetic	
	addition and subtraction	Addition and subtraction have the same priority
LOWEST		

Remarks

Referring to the preceding example, we may now say that 3*L - 6*W means (3*L) - (6*W).

For operators with the same priority (e.g./and *) , operations are carried out from left to right. Thus, 9/3*3 is the equivalent of (9/3)*3 yielding a result of 9.

Using Parentheses to Change Priority

There are times when you will want to change the normal priority of operations. To do this you use pairs of parentheses, exactly as you would in mathematics. When parentheses are used, the operations within the innermost pair of parentheses are performed first, followed by operations within the second innermost pair, and so forth. Within a given pair of parentheses, the normal priority of operations apply.

A simple example of the use of parentheses follows.

Suppose you want to compute $(5X)^2$. If you enter this expression in BASIC as 5*X∧2, first X is squared, then the result is multiplied by 5 because exponentiation has a higher priority than multiplication. To change this, simply enter the expression as (5*X)∧2. In this case; first X is multiplied by 5, then the result is squared.

The more complicated a numeric expression is, the more complicated its BASIC equivalent will be. In the following examples, numeric expressions are shown with their BASIC equivalents. The examples should help you get a better feel for the rules of priority:

EXPRESSIONS

Examples

NUMERIC EXPRESSION	BASIC EQUIVALENT	INTERPRETATION
$\frac{x + y + z}{2}$	(X+Y+Z)/2	1. Add X, Y and Z 2. Divide the sum by 2
$x + \frac{y + z}{2}$	X+(Y+Z)/2	1. Add Y to Z 2. Divide the sum by 2 3. Add X to the result
$2x + 5$	2*X+5	1. Multiply X by 2 2. Add 5 to the result
$2(x + 4)$	2*(X+4)	1. Add 4 to X 2. Multiply the sum by 2
$x^2 + 3$	X∧2+3	1. Square X 2. Add 3 to the result
$(x + 3)^2$	(X+3)∧2	1. Add 3 to X 2. Square the result
$\frac{(x + 3)^2}{4}$	(X+3)∧2/4	1. Add 3 to X 2. Square the sum 3. Divide the result by 4
$\frac{x^2}{6} \cdot \frac{x + y}{2}$	(X∧2/6)*((X+Y)/2))	1. Square X and divide by 6 2. Add X to Y and divide by 2 3. Multiply the two results by each other

Remarks

It is good programming practice to use parentheses whenever you doubt the clarity of an expression, even when they are not strictly necessary.

The expressions used in your program can get very complex. If you save a program and do not run it often, you can easily forget exactly what computations are being performed. For this reason, you may find it useful to put descriptive remarks in a program as you write it. BASIC provides the REM statement and the comment fields specifically for this purpose.

Type of Expression

The type of a numeric expression, i.e. the data-type of the result of the evaluation of an expression (before assigning it to a variable) depends on the type of its operands.

There are four different situations depending on the type of the two operands involved. If the expression involves more than two operands, it can be considered as a series of calculations involving two operands.

The table below summarises the four possible situations.

IF...	THEN...	DISPLAY
both operands are of the same numeric type (integer, single-precision or double-precision	the result is also of that type	A# = 3.29745219 Ok B# = 4.5729719D-1 Ok ?A# +B# 3.75474938 Ok
one operand is integer and the other is single-precision	the result is single-precision	I% = 25 Ok C! = 4.2975 Ok ?I%-C! 2Ø.7Ø25 Ok
one operand is integer and the other is double-precision	the result is double-precision	?I%*A# 82.43630475 Ok
one operand is single-precision and the other is double-precision	the result is double-precision	?C!/B# 9.39760887993736 Ok

Rounding, Overflow and Underflow

Floating point types are forms of approximation to the real numbers of mathematics.

IF...	THEN...
one or more operands in a numeric expression are floating point	calculations are approximate and accuracy can be lost. If this happens the less significant digits are lost and the last digit maintained is rounded off
the value of the expression is bigger than the maximum length allowed for that data-type	an "Overflow" error message is displayed, machine infinity* with the algebraically correct sign is supplied as the result, and execution continues
a division by zero is encountered	the "Division by zero" error message is displayed, machine infinity* with the sign of the numerator is supplied as the result of the division, and execution continues
the evaluation of an exponentiation results in zero being raised to a negative power	the "Division by zero" error message is displayed, positive machine infinity* is supplied as the result of the exponentiation, and execution continues
the value of the expression is smaller than the smallest representable value	the value becomes zero (Underflow) and execution continues
in a numeric assignment, the type of the expression is different from the type of the receiving variable	the expression is automatically converted to the type of the receiving variable

Note: Machine infinity is displayed as 3.4Ø282E+38.

Undefined Values

If a numeric variable in a numeric expression has not yet been set, it is set to zero.

Undetermined Forms

The evaluation of a numeric expression may result in an undetermined form, such as:

Ø/Ø: the message "Division by zero" is displayed and the value 3.4Ø282E+38 (machine infinity) is supplied
ØˆØ: the value is assumed to be 1.

STRING EXPRESSIONS

BASIC permits the use of string expressions, similar in many ways to the numeric expressions we have just looked at. A string expression can be either a string constant, a single string variable, a string array element, a string function, or a mixture of them linked by plus signs (+).

By using the plus sign, strings can be joined - "concatenated" is the technical term. These are some examples of string expressions in LET statements:

```
 5Ø LET A$ = "Chicago,"
 9Ø B$ = "IL.,"
1ØØ N$ = A$+B$+"USA"
```

The concatenation in statement 1ØØ would result in N$ being assigned the string:

Chicago,IL.,USA

When two or more strings are concatenated, the length of the resulting string is the sum of the individual strings. The expression evaluation proceeds from left to right.

Be careful not to assign more than 255 characters to a string variable. In this case, the system issues an error message:

String too long

Remark

A string operand appearing in a string expression may be the null string (""). The null string will also be the default value of a non-initialized string variable.

RELATIONAL EXPRESSIONS

Relational expressions compare either two numeric or two string expressions by means of a relational operator.

Relational Operators

The relational operators are:

=	equals (the equals sign is also used to assign a value to a variable, see LET statement)
>	greater than
<	less than
>= or =>	greater than or equal to
<= or =<	less than or equal to
<> or ><	not equal to

It is illegal to compare a numeric expression with a string expression and vice versa. For example:

A + B>C	is valid
C + D>= E + F	is valid
A$ = B$	is valid
B$ >C1	is wrong if C1 is a numeric variable.

Comparison of numbers has an obvious meaning. Character strings, may also be compared, with the outcome dependent on the numeric value of the character's representation. (This is taken to be the decimal ASCII value of each character within the string). String scanning is performed from left to right, character by character and ends when the first pair of different characters is encountered. The result of the comparison is made on the basis of the first pair of different characters.

Numeric or string expressions are performed first, then relational operators are applied to the result of such expressions.

For example, to write

A>B + C

and

A>(B + C)

is equivalent.

The result of a relational expression is numeric. It is displayed as either -1 (if the relation is true) or Ø (if it is false).

Examples

Let us look at some examples using relational expressions. First let us assign values to the variables X and Y.

DISPLAY	COMMENTS
X=1 Ok Y=2 Ok	BASIC executes the specified assignments
?X>Y Ø Ok	BASIC displays Ø (i.e. false), as X is not greater than Y
?X<>Y -1 Ok	BASIC displays -1 (i.e. true), as X is different from Y
?SIN (X)<Ø Ø Ok	BASIC displays Ø (i.e. false), as SIN(X) is positive
?X MOD Y=1 -1 Ok	BASIC displays -1 (i.e. true), as X MOD Y equals 1

```
?"TOKYO">"FRANKFURT"
-1
Ok
```

BASIC displays -1 (i.e. true) as TOKYO is greater than FRANKFURT (i.e. it comes after FRANKFURT in alphabetical order)

```
?"TOKYO">"TOKYO1"
 Ø
Ok
```

BASIC displays Ø (i.e. false) as TOKYO is less than TOKYO1. Where two strings are of unequal length and the shorter string exactly matches the first part of the larger string then the longer string is considered greater than the shorter one

Using Relational Expressions

The result of a relational expression may be used to make a decision regarding program flow. You can use relational expressions in the following control statements:

- IF... GOTO... ELSE, or
- IF... THEN... ELSE, or
- WHILE

where a condition is tested to determine later operations in the program (see Chapter 8).

The condition may be a numeric, relational or logical expression. BASIC determines whether the condition (after IF or WHILE) is true or false by testing the result of the expression for non-zero and zero respectively. A non-zero result is assumed to be true, and a zero result is false.

For example, the following statement:

```
1ØØ IF A$>B$ THEN 5Ø
```

will transfer control of execution to statement 5Ø if the condition (A$>B$) is true, (i.e. A$ greater than B$). If the condition is false (i.e. A$ not greater than B$) the next statement will be executed.

LOGICAL EXPRESSIONS

A logical expression consists of one operand preceded by the logical operator NOT, two operands separated by another logical operator (AND, OR, XOR, EQV and IMP), or two operands separated by a logical operator and NOT.

The operands in a logical expression may be numeric or relational expressions. Both have numeric values.

The result of a logical expression is also numeric: it is an integer value with any combination of bits in the range -32768 to 32767.

Examples of logical expressions are:

NOT X	is valid
X AND Y	is valid
A>B OR C>D	is valid
I% AND A$<B$	is valid
A$ XOR B$	is not valid (as the operands are string)

Logical Operators

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range -32768 to +32767. (If the operands are not in this range, an error occurs.) The given operation is performed on these integers, each bit of the result being determined by the corresponding bits in the two operands.

The logical operators are listed below in a table called the "truth table". It describes graphically the results of the logical operations on a bit-by-bit basis. Every possible combination of bits is given. (Notice that the two operators XOR and EQV are exact opposites.)

A	NOT A	A	B	A AND B	A OR B	A XOR B	A EQV B	A IMP B
1	Ø	1	1	1	1	Ø	1	1
Ø	1	1	Ø	Ø	1	1	Ø	Ø
		Ø	1	Ø	1	1	Ø	1
		Ø	Ø	Ø	Ø	Ø	1	1

Table 6-1 The Truth Table

Logical Operator Priority

In an expression, logical operations are performed after numeric and relational operations.

The table below lists logical operators in the order BASIC evaluates them.

OPERATORS	PRIORITY
NOT	HIGHEST
AND	
OR	
XOR	
IMP	
EQV	LOWEST

Table 6-2 Logical Operator Priority

Examples

Let us look at some examples. First let us assign values to the variables X, Y, AND Z.

DISPLAY	COMMENTS
X%=Ø Ok Y%=3 Ok Z%=5 Ok	BASIC executes the specified assignments
?X%<Y% AND Z%=3 Ø Ok	the result is false (Ø), as X%<Y% is true (-1) but Z%=3 is false (Ø)
?X% OR X%<Z% -1 Ok	the result is true (-1), as X% is false (Ø) but X%<Z% is true (-1)

```
?63 AND 16
 16
Ok
```

the result is 16, as

63 = binary 111111
16 = binary Ø1ØØØØ
Ø1ØØØØ

```
?4 OR 2
 6
Ok
```

the result is 6, as

4 = binary 1ØØ
2 = binary Ø1Ø
11Ø

```
?-1 OR-2
-1
Ok
```

the result is -1, as

-1 = binary 1111111111111111
-2 = binary 1111111111111110
1111111111111111

```
?Ø<2 AND 4=4
-1
Ok
```

the result is true (-1), as

Ø<2 is true (-1), and
4=4 is true (-1)

```
?Ø XOR Y%=3
-1
Ok
```

the result is true (-1), as

Ø is false (Ø), and
Y%=3 is true (-1)

```
?Z%>Y% AND NOT "A">"B"
-1
Ok
```

the result is true (-1), as

Z%>Y% is true (-1), and
"A">"B" is false (Ø)

Note: It is possible to write two consecutive logical operators only if the second one is the NOT operator

Using Logical Expressions

You can use logical expressions:

- to test a condition in the following control statements:

IF... GOTO... ELSE,
IF... THEN... ELSE,
WHILE

For example:

5Ø IF A$>B$ and B<=C THEN 3ØØ

will transfer control of execution to statement 3ØØ if the condition (A$>B$ AND B<=C) is true (i.e. A$ greater than B$ and B less than or equal to C). If the condition is false (i.e. A$ less than B$ or B greater than C) the next statement will be executed.

- to test words (16 bits) for a particular bit pattern. For example the AND operator may be used to "mask" all but one of the bits of a status word at a machine I/O port. The OR operator may be used to "merge" two words to create a particular binary value.

For example:

-1 AND 8 is 8
and:
-1 OR 8 is -1
as
-1 = binary 1111111111111111
 8 = binary ØØØØØØØØØØØØ1ØØØ

OPERATOR PRIORITY

The table below lists all operators (numeric, string, relational, and logical) in the order BASIC evaluates them.

OPERATORS		PRIORITY
^	(exponentiation)	HIGHEST
-	(negation)	
* /	(multiplication and division)	
\	(integer division)	
MOD	(modulus arithmetic)	

+	-	(addition and subtraction)
	+	(string concatenation)
All relational operators		
NOT		
AND		
OR		
XOR		
IMP		
EQV		LOWEST

Table 6-3 Operator Priority

Remarks

- operators shown on the same line have equal precedence
- all relational operators have equal precedence
- evaluation order of expressions can be overridden by the use of parentheses. For example the evaluation order of:

  NOT A>B AND C>D OR E>F

  is different from the evaluation order of:

  NOT (A>B AND (C>D OR E>F))

  if, for instance, A>B is true, C>D is false and E > F is true, the first expression is true, whereas the second is false.
- the result of any expression can also be an operand, thus you can form very complex expressions, for instance chaining two or more logical expressions by a logical operator (as in the examples above). However it is not good programming practice to write too complex expressions.

7. HOW BASIC OUTPUTS DATA

ABOUT THIS CHAPTER

You have now seen how to input data to the M20 and how to process it.

In this chapter you will see how to set the screen or printer line width (WIDTH command) and how to get results from the computer. We shall examine the LPRINT, PRINT, LPRINT USING and PRINT USING statements. They allow you to output data either in a standard or in a user-defined format.

CONTENTS

SETTING THE NUMBER OF NULLS AND THE WIDTH

The NULL command (which may also be used in a program) allows you to set the number of nulls printed after each line.

The WIDTH command (which may also be used in a program) allows you to set the screen or printer line width.

NULL (PROGRAM/IMMEDIATE)

Sets the number of nulls to be printed at the end of each line and hence delays the printing of the next line.

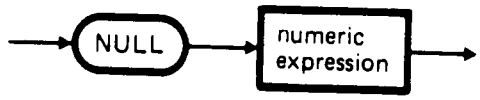


Figure 7-1 NULL Command

Example

IF you enter...	THEN...
NULL 2 CR	2 nulls will be printed after each line. Note: The numeric expression is rounded to the nearest integer (if necessary). For 1Ø-character-per-second tape punches the numeric expression value should be >= 3. This also identifies lines on the tape. When tapes are not being punched, this value should be Ø or 1 for teletypes and teletype-compatible CRTs. This value should be 2 or 3 for 3Ø cps hard copy printers

WIDTH (PROGRAM/IMMEDIATE)

Sets screen or printer line width, when a PRINT, WRITE, LPRINT, PRINT USING, LPRINT USING statement is executed or an error message is issued.

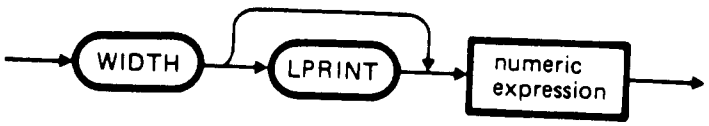


Figure 7-2 WIDTH Command

Default Values

If you do not use a WIDTH command a screen width of 64 characters is assumed.

If you do not use a WIDTH LPRINT command a printer line width of 132 characters is assumed.

Examples

```
1Ø PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RUN
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ok
WIDTH 18
Ok
RUN
ABCDEFGHIJKLMNOPQR
STUVWXYZ
Ok
```

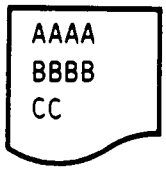
Characteristics

IF...	THEN...
the LPRINT option is omitted	the line width is set for the screen only

LPRINT is included — the line width is set for the line printer.

For example:

```
10 WIDTH LPRINT 4
20 LPRINT "AAAABBBBCC"
RUN
Ok
```



the numeric expression value is not an integer — it is rounded to the nearest integer and must have a value in the range 15 to 255.

If the rounded value is 255, the line width is "infinite", that is, BASIC never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255.

If the rounded value is greater than 255 an error is issued. (Illegal function call.)

STANDARD FORMAT

You may output your results in standard format by the PRINT, WRITE and LPRINT statements. They can be used as immediate statements too. They allow you to have the results of calculations either printed (LPRINT) or displayed (PRINT and WRITE).

If you wish to output the results of two or more expressions on one line, separate your expressions with commas (WRITE) and with commas or semicolons (PRINT, LPRINT).

With the WRITE statement each item displayed will be separated from the last by a comma (and strings will be delimited by quotation marks). With the PRINT and LPRINT statements, if you use commas, the results will be separated, whereas semicolons will cause the results to be packed together and the strings will not be delimited by quotation marks.

LPRINT/PRINT (PROGRAM/IMMEDIATE)

LPRINT prints a list of data in a standard format.

PRINT displays a list of data in a standard format. A question mark (?) may be used instead of PRINT.

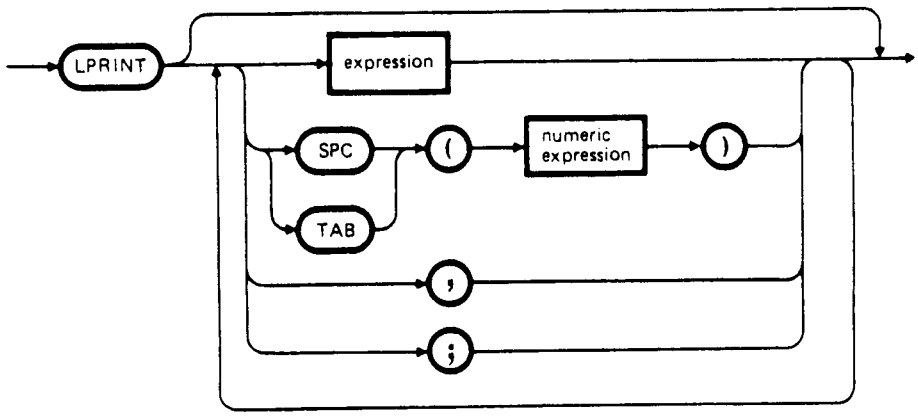


Figure 7-3 LPRINT Statement

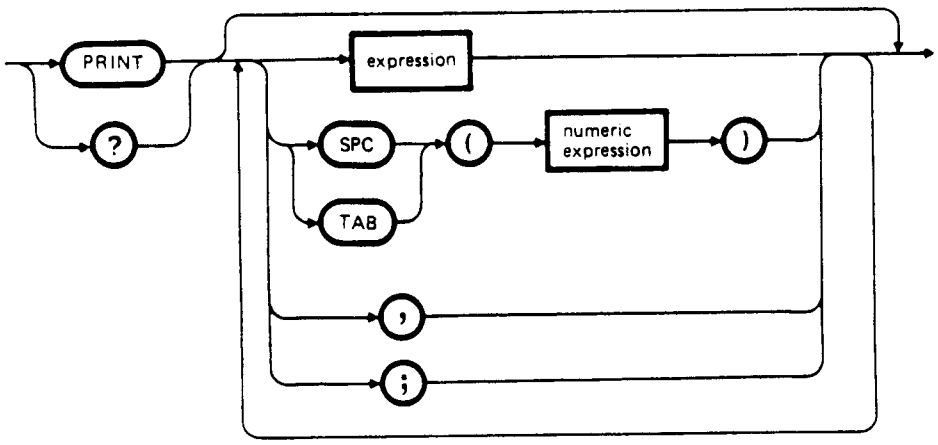


Figure 7-4 PRINT Statement

HOW BASIC OUTPUTS DATA

Characteristics

IF...	THEN...
an LPRINT (or a PRINT) statement does not end with a comma or semi-colon	a new line of output is printed (or displayed) when the statement is executed. For example: LIST 1Ø PRINT 1 2Ø PRINT 2 Ok RUN 1 2 Ok
no expressions appear in an LPRINT (or a PRINT) statement	a line is skipped. For example: LIST 1Ø PRINT 1 2Ø PRINT 3Ø PRINT 2 Ok RUN 1 2 Ok This form of LPRINT (or PRINT) is useful for producing spaces between output lines
expressions in the output list are separated by commas	each value is printed (or displayed) left justified in one of the "print zones" in which each line is divided. (Each zone has 14 positions).

For example:

```
LIST
1Ø A$ = "For June..."
2Ø X = .353
3Ø PRINT "Results", A$, X
Ok
RUN
Results        For June...     .353
Ok
```

Note: Each positive value (in this case .353) is preceded by a space (see below).

The number of print zones on each line depends on the maximum number of characters each can contain. This may be specified by the WIDTH command or assumed by default.

String values displayed (or printed with LPRINT) are not delimited by quotation marks

the list of expressions has many entries

two or more lines of output may be produced.

For example:

```
WIDTH 31
Ok
PRINT 1, 1+2, 2+3, 7, 9, "ABCD"
 1             3
 5             7
 9            ABCD
Ok
```

Note: Each positive value is preceded by a space (see below)

one or more numeric expressions appear in an LPRINT (or a PRINT) statement

- each value printed or displayed is always followed by a space
- each positive value is preceded by a space
- each negative value is preceded by a minus sign

- each single-precision value that can be represented with 6 or fewer digits in the fixed point format as accurately as it can be represented in the floating point (or "exponential") format, is output using the fixed point format

- each double-precision value that can be represented with 15 or fewer digits in the fixed point format as accurately as it can be represented in the floating point (or "exponential") format, is output using the fixed point format:

  For example:

```
PRINT 1Ø∧-6
 .ØØØØØ1
Ok
PRINT 1Ø∧-7
 1E-Ø7
Ok
PRINT 1D-15, 1D-16
 .ØØØØØØØØØØØØØØ1            1D-16
Ok
```

Note: The second value is displayed left justified in the third print zone (as the first value overflows into the second print zone)

a comma follows the last expression in the list

the next character or digit issued as output (that is, the first character or digit in a subsequent PRINT or INPUT or LPRINT operation) is printed or displayed on the same line (at the beginning of the next print zone) if sufficient space is available (otherwise on a new line).

For example:

```
LIST
1Ø A$ = "For July..."
2Ø X = .491
3Ø PRINT "Results", A$,
4Ø PRINT X
Ok
RUN
Results       For July...     .491
```

a semicolon follows the last expression in the list

the next character or digit issued as output (that is, the first character or digit in a subsequent PRINT, or INPUT or LPRINT operation) is printed or displayed on the same line (at the cursor position) - if sufficient space is available, otherwise on a new line.

For example:

```
LIST
1Ø INPUT X
2Ø PRINT X "SQUARED IS" X∧2 "AND";
3Ø PRINT X "CUBED IS" X∧3
4Ø PRINT
5Ø GOTO 1Ø
Ok
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261
?
```

commas are used consecutively

the effect of each comma is to position the print head (or the cursor) at the start of the next zone.

The use of commas in this way lets you display (or print) data widely spaced.

For example:

```
PRINT "M",,"N"
M                              N
Ok
```

semicolons or blanks are used instead of commas to separate expressions in the list

output values are spaced more closely. The exact spacing depends on the number of digits or characters in each value. The use of semicolons in this way allows you to print (or display) more values on each line.

Having more than one space or semicolon between expressions has the same effect as one space or semicolon.

For example:

```
LIST
10 A1 = 1000
20 A2 = 2000
30 A3 = 3000
40 A4 = 4000
50 A5 = 5000
60 A6 = 6000
70 A7 = -7000
80 PRINT A1;A2;A3;A4;;A5 A6 A7
Ok
RUN
1000  2000  3000  4000  5000  6000 -7000
```

The spaces between the numbers appear because the system adds one space after printing (or displaying) each number and eliminates the implied plus sign before each positive value

you mix semicolons and commas in the same LPRINT (or PRINT) statement

you get a simple method of labelling each of your results and of gaining wide spaces within a line.

For example:

```
LIST
1Ø INPUT "Length and Width"; L,W
2Ø PRINT "Area =";L*W,"Length =";L,"Width =";W
3Ø GOTO 1Ø
Ok
RUN
Length and Width? 1.2, 3
Area = 3.6     Length = 1.2    Width = 3
Length and Width? ^C
Break in 1Ø
Ok
```

you use the special built-in function TAB

you name the precise print (or cursor) position, in a line, at which you want your next data item to begin.

For example:

```
PRINT 1; TAB(6); 2
 1    2
Ok
```

you use the special built-in function SPC

you insert a specified number of blanks on the line. (In calculating the number of blanks you want, remember that numeric data is always output with one blank after it).

For example:

```
PRINT 1; SPC(6); 2
 1         2
Ok
```

WRITE (PROGRAM/IMMEDIATE)

Displays a list of data. Each item displayed will be separated from the last by a comma. Strings will be delimited by quotation marks("). After the last item is displayed, BASIC inserts a carriage return/line feed.

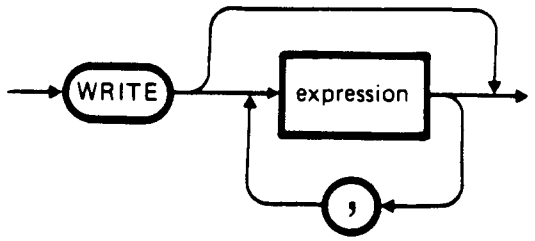


Figure 7-5 WRITE Statement

Where

Expression may be a numeric, relational, logical, or string expression. If no expression is indicated a blank line is output.

Example

DISPLAY	COMMENTS
1Ø A=8Ø : B=9Ø 2Ø C$="THAT'S ALL" 3Ø WRITE A,B,C$ RUN 8Ø, 9Ø, "THAT'S ALL" Ok	when a WRITE statement is executed, each item is separated from the last by a comma, and strings are delimited by quotation marks. Note: Numeric values are displayed using the same format as the PRINT statement but they are not followed by blanks

USER DEFINED FORMAT

You have seen that the use of commas, semicolons, quoted strings, and the SPC and TAB functions provides limited control of the format of displayed or printed information. Two statements, LPRINT USING and PRINT USING, provide the capability of generating printed or displayed output with complete control of the format.

They are usually used in a program, but they can be used as immediate statements too.

LPRINT USING/PRINT USING (PROGRAM/IMMEDIATE)

LPRINT USING prints a list of data in a user-defined format.

PRINT USING displays a list of data in a user-defined format.

The expressions appearing in an LPRINT (or PRINT) USING statement must be separated by commas (,) or semicolons (;), it makes no difference which punctuation mark is used. Values will be output (printed or displayed) in a format specified by the string expression appearing after USING. This expression is a string literal or variable that is composed of special formatting characters. These formatting characters (see below) determine the fields and the format of the output strings or numbers.

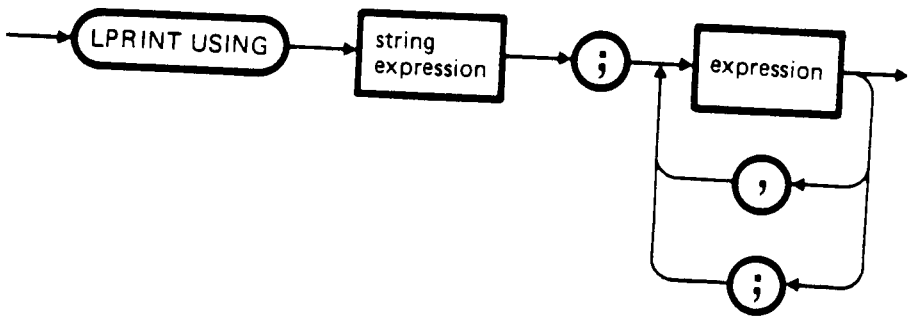


Figure 7-6 LPRINT USING Statement

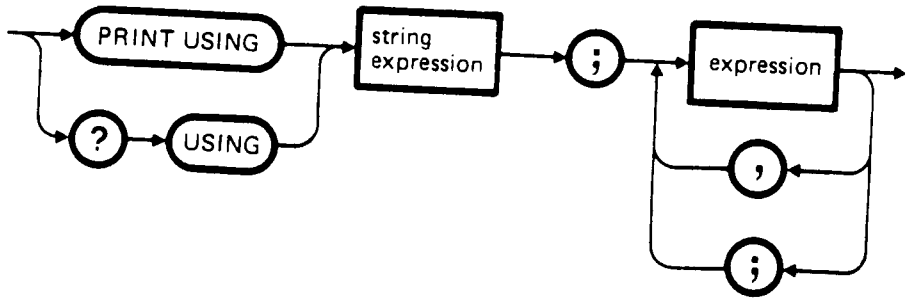


Figure 7-7 PRINT USING Statement

Where

SYNTAX ELEMENT	MEANING
string expression	is a string of formatting characters (see below) or a string variable consisting of a string of formatting characters
expression	is a numeric, relational, logical or string expression which is to be printed or displayed

To Output Strings

One of three formatting characters may be used:

FORMATTING CHARACTER	MEANING
"!"	specifies that only the first character in the given string is to be output. For example: LIST 1Ø A$="WATCH" 2Ø B$="OUT" 3Ø PRINT USING "!";A$;B$ Ok RUN WO Ok
"\ n spaces \"	specifies that 2+n characters from the string are to be output. If the backslashes are entered with no spaces, two characters will be output. With one space, three characters will be output, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces to the right.

For example:

```
LIST
1Ø A$="LOOK"
2Ø B$="OUT"
3Ø PRINT USING "\   \";A$;B$
4Ø PRINT USING "\    \";A$;B$
Ok
RUN
LOOKOUT
LOOK OUT
```

"&" specifies a variable length string field. When the field is specified with "&", the string is output exactly as input.

Example:

```
LIST
1Ø A$="LOOK":B$="OUT"
2Ø PRINT USING "!";A$;
3Ø PRINT USING "&";B$
Ok
RUN
LOUT
Ok
```

To Output Numbers

The following formatting characters may be used:

FORMATTING CHARACTERS	MEANING
#	a number sign is used to represent each digit position. Digit positions are always filled. If the number to be output has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

For example:

```
PRINT USING "####";99 CR
  99
Ok
```

a decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will be output only if it is different from zero. Numbers are rounded when necessary.

For example:

```
PRINT USING "###.##";.78 CR
   .78
Ok

PRINT USING "###.##";987.654 CR
987.65
Ok

PRINT USING "##.##    ";1Ø.,5.3,66.789,.234 CR
1Ø.ØØ    5.3Ø    66.79      .23
Ok
```

In the last example, three spaces were inserted at the end of the format string to separate the displayed values on the line.

\+

a plus sign at the beginning or end of the format string will cause the number sign (plus or minus) to be output before or after the number.

For example:

```
PRINT USING "+###.##    ";-68.95,2.4,55.6,-.9 CR
-68.95    +2.4Ø    +55.6Ø      -.9Ø
Ok
```

Note: If you only want the minus sign (not the plus sign) to precede the number, you should start the format string with an extra number sign (#).

For example:

```
PRINT USING "###.##";-68.95,68.95 CR
-68.95 68.95
```

\-

a minus sign at the end of the format field will cause negative numbers to be output with a trailing minus sign.

For example:

```
PRINT USING "##.##-    ";-68.95,22.449,-7.Ø1 CR
68.95-   22.45        7.Ø1-
Ok
```

**

a double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

For example:

```
PRINT USING "**#.#    ";12.39,-Ø.9,765.1 CR
*12.4   **-.9    765.1
Ok
```

$$

a double dollar sign causes a dollar sign to be output to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format (^^^^) cannot be used with $$. Negative numbers cannot be used unless the format string ends with a minus or a plus sign. In the former case negative numbers appear with the negative sign on the right, in the latter case both positive and negative numbers appear with the appropriate sign on the right.

For example:

```
PRINT USING "$$###.## -";-456.78 CR
 $456.78-
Ok
```

**$

the **$ at the beginning of a format string combines the effects of the two symbols described above. Leading spaces will be asterisk-filled

and a dollar sign will be inserted before the number. **$ specifies three more digit positions, one of which is the dollar sign.

For example:

```
PRINT USING "**$##.##";2.34 CR
***$2.34
Ok
```

,

A comma to the left of the decimal point in a formatting string causes a comma to be output to the left of every third digit to the left of the decimal point. A comma at the end of the format string is output as part of the string. A comma specifies another digit position. A comma has no effect if used with the exponential (^^^^) format.

For example:

```
PRINT USING "####,.##";1234.5 CR
1,234.5Ø
Ok

PRINT USING "####.##,";1234.5 CR
1234.5Ø,
Ok
```

^^^^

four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be output. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to output a space or a minus sign.

For example:

```
PRINT USING "##.##^^^^";234.56 CR
 2.35E+Ø2
Ok
```

```
PRINT USING ".####^^^^";888888 CR
.8889E+Ø6
Ok
PRINT USING "+.####^^^^";123 CR
+.123ØE+Ø3
Ok
```

_ an underscore in the format string causes the next character to be output as a literal character (i.e. as it appears in the format string)

For example:

```
PRINT USING "_!##.##_!";12.34 CR
!12.34!
Ok
```

The literal character itself may be an underscore by placing "_ _" in the format string

% if the number to be output is larger than the specified numeric field, a percent sign is output in front of the number. If rounding causes the number to exceed the field, a percent sign will be output in front of the rounded number.

For example:

```
PRINT USING "##.##";111.22 CR
%111.22
Ok

PRINT USING ".##";.999 CR
%1.ØØ
Ok
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result

Remarks

If the same format string is to be used several times in a program, you may find it convenient to assign the formatting characters to a string variable and then specify the variable name instead of the format string. This technique is shown below:

```
1Ø A$="##.##"
2Ø PRINT USING A$; 8.49
 .
 .
 .
1ØØ PRINT USING A$;A,B,C
 .
 .
 .
15Ø PRINT USING A$;A1,B1
 .
 .
 .
RUN
 8.49
 Ø.ØØ Ø.ØØ Ø.ØØ
 Ø.ØØ Ø.ØØ
Ok
```

8. CONTROL STATEMENTS

ABOUT THIS CHAPTER

Normally the statements in a BASIC program are executed sequentially in the same order as they appear, following the line numbers of the statements. Sometimes, however, it is necessary to "branch" to some other part of the program, thus changing the normal sequence of execution.

Branches and loops are two methods of altering the normal flow of program execution. In this chapter we shall examine conditional and unconditional branches as well as loops.

CONTENTS

CONTROL STATEMENTS

UNCONDITIONAL BRANCHES

Branches may be conditional or unconditional. The GOTO statement causes an unconditional transfer of control. In the statement, you simply indicate the line number to which control is to be transferred. The sample program, RECTANGLE1 (see Chapter 1 and 2) contains the following GOTO statement:

```
8Ø GOTO 2Ø
```

This statement tells the M20 to execute statement 20 next, rather than the statement with the next higher line number.

There is one more form of unconditional branching; the ON...GOTO or computed GOTO statement. This enables you to transfer control to one of perhaps several statements, depending on the value of a numeric expression. For example:

```
1ØØ ON A GOTO 15, 3Ø, 5ØØ
```

This statement says; if A=1, go to statement 15, if A=2 go to statement 3Ø, if A=3 go to statement 5ØØ but if A < 1 or A > 3, BASIC continues with the next executable statement.

GOTO (PROGRAM/IMMEDIATE)

Transfers control to a specified program line.

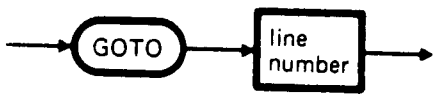


Figure 8-1 GOTO Statement

Examples

DISPLAY

```
LIST
1Ø READ R
2Ø PRINT "R =";R,
3Ø A = 3.14*R ∧ 2
4Ø PRINT "AREA =";A
5Ø GOTO 1Ø
6Ø DATA 5,7,12
Ok
RUN
R = 5           AREA = 78.5
R = 7           AREA = 153.86
R = 12          AREA = 452.16
Out of data in 1Ø
Ok
```

COMMENT

statement 5Ø transfers control unconditionally to statement 1Ø

Characteristics

IF...	THEN...
you enter GOTO 5ØØ **CR** when you are in Command Mode AND 5ØØ is a statement of your current program	GOTO is used as an alternative to RUN. Note: GOTO line number used in Command Mode causes execution to begin at the specified line number without an automatic CLEAR. This lets you pass values to program variables while in Command Mode. This technique may be used in debugging your program
the statement specified by line number is non executable (e.g. a REM statement)	control is passed to the first subsequent executable statement

IF...	THEN...
a GOTO statement is encountered within a FOR/NEXT loop AND transfers control outside the loop	the value of the control variable (see FOR/NEXT below) is the last value assumed within the loop

ON...GOTO (PROGRAM/IMMEDIATE)

Transfers control to one of several specified lines, according to the value of a specified expression.

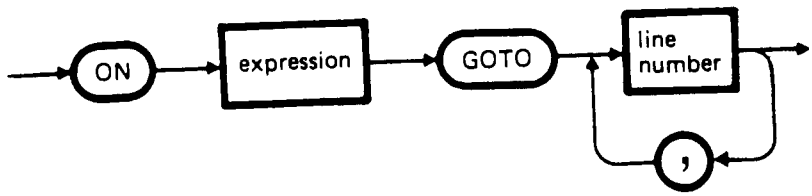


Figure 8-2 ON...GOTO Statement

Characteristics

IF...	THEN...
you have a program so structured: 2Ø INPUT A 3Ø ON A GOTO 1ØØ,2ØØ,3ØØ 4Ø ... 9Ø 1ØØ ... 2ØØ ... 3ØØ A=1 A=2	the value of A determines which line number in the list will be used for branching. For example, if the value is 3, the third line number in the list will be the destination of the branch. (If the value of A is a non-integer, the fractional portion is rounded)

the value of the expression after ON is zero or greater than the number of items in the list (but less than or equal to 255)	BASIC continues with the next executable statement
the value of the expression after ON is negative or greater than 255	an "Illegal function call" error occurs

CONDITIONAL BRANCHES

In many situations you will want to branch to different portions of a program depending on conditions that arise within it. To test these conditions and make a decision as to what to do next, you can use an IF...GOTO...ELSE, or an IF..THEN...ELSE statement.

IF...GOTO...ELSE/IF...THEN...ELSE (PROGRAM/IMMEDIATE)

Both these statements transfer control, conditionally, to a specified statement.

IF...THEN...ELSE is more powerful (as you can see by the syntax); it allows a series of statements to be entered both after THEN and ELSE.

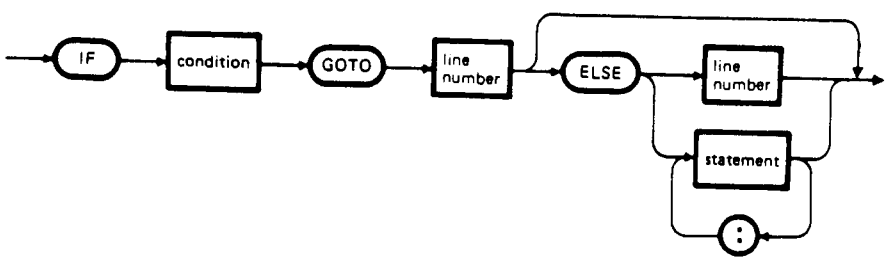


Figure 8-3 IF...GOTO...ELSE Statement

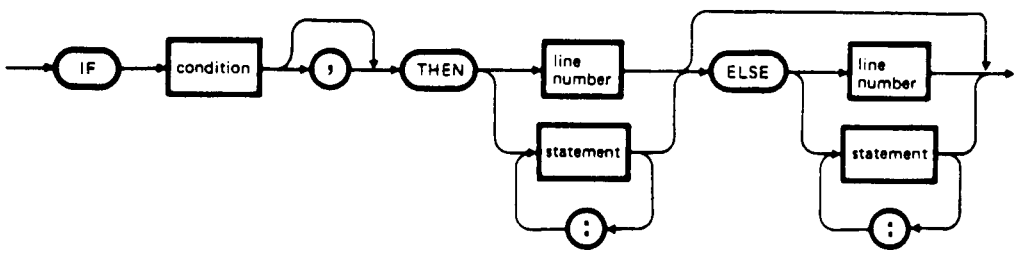


Figure 8-4 IF...THEN...ELSE Statement

Where

SYNTAX ELEMENT	MEANING
condition	may be: - a numeric expression - a relational expression - a logical expression. Note: BASIC determines whether the condition is true or false by testing the result of the expression for non zero and zero respectively. A non zero result is true and a zero result is false. Because of this, you can test whether the value of a variable is non zero or zero by merely specifying the name of the variable as "condition". A comma is allowed before THEN

Characteristics

IF...	THEN...
the condition is true	control is EITHER passed to the statement whose line number is specified after GOTO (or THEN) OR to the first statement specified after THEN
the condition is false AND IF the ELSE clause is omitted	control is passed to the next executable statement following the IF...GOTO or IF...THEN statement
the condition is false AND IF the ELSE clause is present	control is EITHER passed to the statement whose line number is specified after ELSE OR to the first statement specified after ELSE. Note: After executing the statement(s) following ELSE, control is passed to the next executable statement

Examples

DISPLAY	COMMENTS
LIST 1Ø REM IF GOTO test program 2Ø INPUT X% 3Ø IF X%>=1Ø GOTO 6Ø 4Ø PRINT "IF GOTO failed the test" 5Ø GOTO 99 6Ø PRINT "IF GOTO passed the test" 99 GOTO 2Ø Ok RUN ? 1Ø IF GOTO passed the test	if you enter 1Ø the condition (X%>=1Ø) in statement 3Ø is true and control is transferred to statement 6Ø. If you enter 2 the condition is false and control is transferred to statement 4Ø

```
? 2
IF GOTO failed the test
? ^C
Break in 20
Ok
```

```
LIST
10 INPUT X
20 IF X=INT(X)
   THEN PRINT X; "is an integer"
   ELSE PRINT X; "is not an integer"
30 IF X=9999 THEN END ELSE 10
Ok
RUN
? 1
 1 is an integer
? 1.5
 1.5 is not an integer
? ^C
Break in 10
Ok
```

if you enter 1, the condition (X=INT(X)) in statement 20 is true and control is transferred to the PRINT statement after THEN. If you enter 1.5 the condition is false and control is transferred to the PRINT statement after ELSE

Note: Statement 20 is one logical line divided into three physical lines.

```
50 IF I THEN A=1000
```

the value 1000 is assigned to variable A if I is not zero

```
70 IF (I<30) AND (I>5) THEN
   A=B+C:GOTO 350
80 PRINT "OUT OF RANGE"
      :
      :
```

a test determines if I is greater than 5 and less than 30. If I is in this range, A is calculated and execution branches to line 350. If I is not in this range execution continues with line 80

Nesting of IF Statements

IF...GOTO...ELSE or IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example:

IF...

THEN...

you enter:

```
IF X>Y  THEN PRINT "GREATER"
ELSE IF Y>X THEN PRINT "LESS THAN"
ELSE PRINT "EQUAL" CR
```

you have entered a legal statement (it is one logical line divided into three physical lines)

the statement does not contain the same number of ELSE and THEN clauses	each ELSE is matched with the most recent unmatched THEN. For example: 1ØØ IF A=B THEN IF B=C THEN PRINT "A=C" ELSE PRINT "A< >C" CR 11Ø... Will display A=C when A=B and B=C; will display A < > C when A=B but B is different from C. If A is different from B control is transferred to the statement 11Ø.

To Test Equality for a Floating Point Value

IF...	THEN...
you use an IF...GOTO...ELSE or an IF...THEN...ELSE statement to test equality for a value that is the result of a floating point computation	the test should be against the range over which the accuracy of the value may vary (as the interval representation of the value may not be exact). For example, to test a computed variable A against the value 1.Ø use: IF ABS(A-1.Ø) 1.ØE-6 GOTO... or IF ABS(A-1.Ø) 1.ØE-6 THEN... This test returns true if the value of A is 1.Ø with a relative error of less than 1.ØE-6

CONTROL STATEMENTS

LOOPS

Repeatedly executing a series of statements is known as looping.

You may create loops by:

- the FOR and NEXT statements; they are used to enclose a series of statements, enabling you to repeat those statements a specified number of times

- the WHILE and WEND statements; they are used to enclose a series of statements, enabling you to repeat those statements as long as a given condition is true.

How a Loop can Simplify Your Task

Suppose you wanted to display a listing of each number from 1 to 25, together with its square root.

You could do it, using the following statements, but this is a very primitive solution to the problem:

```
10 PRINT 1,SQR(1)
20 PRINT 2,SQR(2)
30 PRINT 3,SQR(3)
```

and so on, ending with:

```
240 PRINT 24,SQR(24)
250 PRINT 25,SQR(25)
260 END
```

using an IF...THEN statement instead would be far more efficient:

```
10 LET A=1
20 PRINT A,SQR(A)
30 LET A=A+1
40 IF A<26 THEN 20
50 END
```

A further simplification would be to use a FOR/NEXT loop:

```
10 FOR A=1 TO 25
20 PRINT A,SQR(A)
30 NEXT A
40 END
```

At the moment, this simplification may not seem very dramatic, but the uses to which you can put a FOR/NEXT loop are surprising. We will now explore some of these possibilities.

Starting the Loop - the FOR Statement

The FOR statement identifies the start of a loop; the NEXT statement identifies the end of one. FOR specifies how many times the loop (i.e. the statement or sequence of statements between the FOR and the NEXT statement) is to be executed.

In the preceding example, FOR specifies that the PRINT statement is to be executed for successive values of A from 1 through 25 (an increment of 1 is added to A for each execution of PRINT). When the value of A exceeds 25, execution of the loop stops, and control is passed to the statement following the NEXT statement. In this case, the statement that follows is END, denoting the end of the program.

The specification A=1 TO 25 defines the set of values over which the loop will be executed. In this context, A is called a control variable; controlling the number of times the loop is to be executed. The control variable will always increase by 1 if the FOR statement contains no instructions to the contrary. You can, however, increment the control variable by some value other than 1 if you want to. This is done by adding a STEP clause, for example:

```
1Ø FOR A=1 TO 25 STEP 2
```

This statement indicates an increment (or step) of 2. Thus, the loop will be executed once for every odd value of A from 1 to 25 (that is, 1,3,5,...25). When the value of A exceeds 25 (when it reaches 27), execution of the loop will end. The value of A will be 27 before the statement that follows the NEXT statement is executed.

If you wanted to execute the loop once for every even value of A from 1 to 25, you could specify:

```
1Ø FOR A=2 TO 25 STEP 2
```

Again, when the value of A exceeds 25 (when it reaches 26), execution of the loop will end.

You could explicitly specify a step value of 1, as in the example below:

```
8Ø FOR X=1 TO 4Ø STEP 1
```

but it is unnecessary.

As with the expressions in LET and PRINT statements, specifications in FOR statements can be quite complicated. For example, all of the following FOR statements are valid:

```
7Ø FOR A=B TO C
8Ø FOR X=8/M+N TO A^2
5Ø FOR I=SQR(A) TO 155Ø STEP B*C+6
```

If the value of an increment is negative, the FOR/NEXT loop is executed until the value of control variable is less than the final value (i.e. the value expressed after TO).

For example:

```
LIST
1Ø FOR K%=1 TO -1Ø STEP -1
2Ø PRINT K%;
3Ø NEXT K%
Ok
RUN
 1  Ø -1 -2 -3 -4 -5 -6 -7 -8 -9 -1Ø
```

With this example the loop is repeated 12 times.

Closing the Loop - the NEXT Statement

Just as the loop always begins with a FOR statement, it always ends with a NEXT statement. Remember that the loop comprises all the statements included between the FOR and NEXT statements.

The NEXT statement consists of the keyword NEXT, optionally followed by a list of control variables. Each control variable must be the same as the control variable that appears in the corresponding FOR statement. More than one FOR statement may be associated with only one NEXT statement (see Nested Loops below).

FOR/NEXT (PROGRAM/IMMEDIATE)

FOR and NEXT statements allow a series of statements to be performed in loop a given number of times.

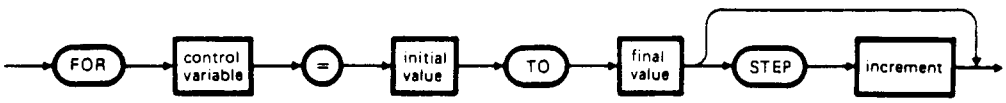


Figure 8-5 FOR Statement

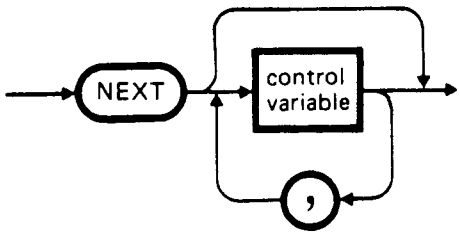


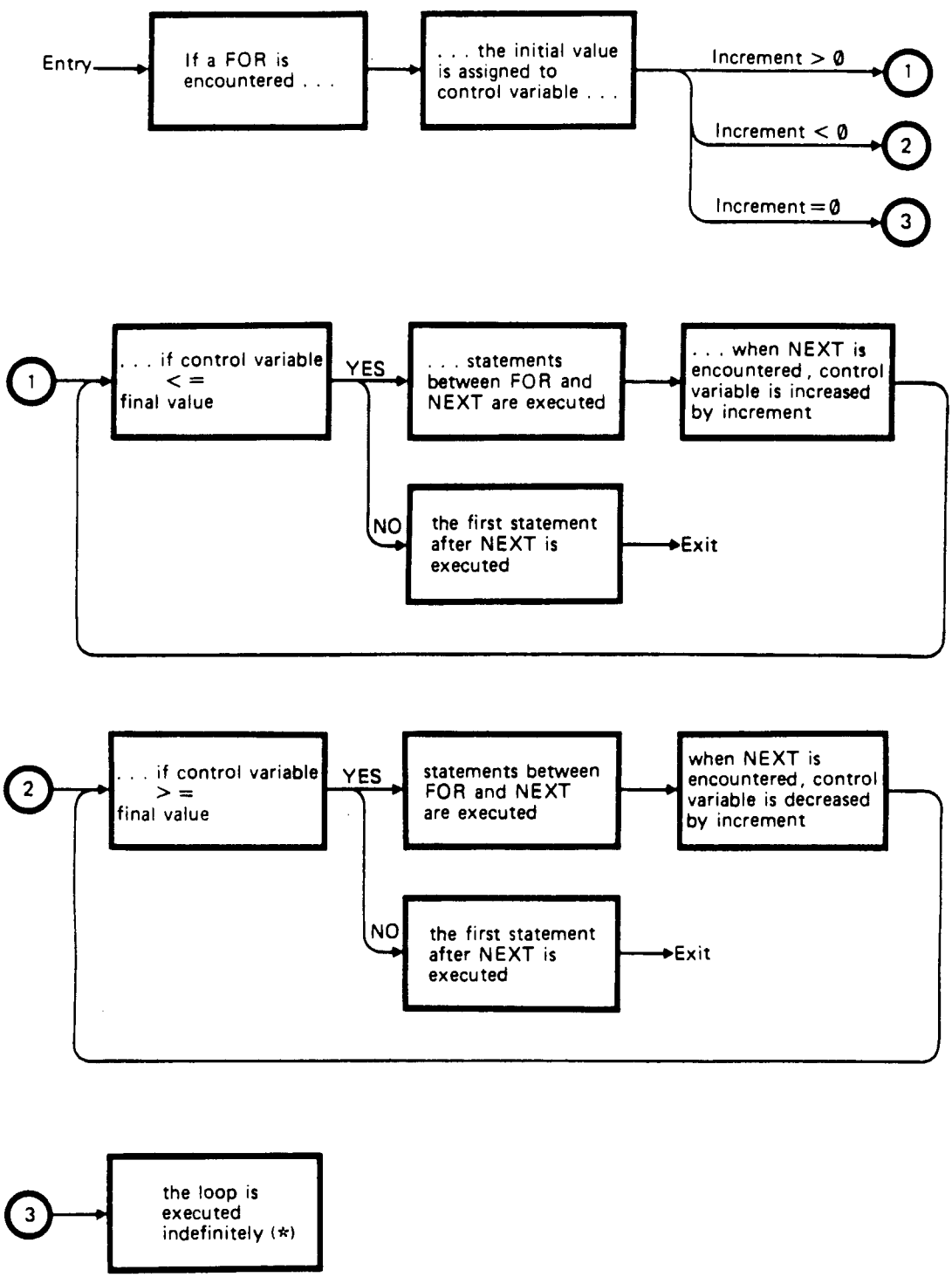
Figure 8-6 NEXT Statement

Where

SYNTAX ELEMENT	MEANING	DEFAULT VALUES
control variable	is a simple numeric variable (defined as an integer or a single-precision variable). The name of the control variable specified in the NEXT statement must be the same as that specified in the FOR statement but the NEXT statement may specify a list of control variables (see Nested Loops below) or even none	if a NEXT statement specifies no control variable the NEXT statement will match the most recent FOR statement

initial value	is a numeric expression specifying the first value assigned to the control variable when the FOR statement is executed	
final value	is a numeric expression specifying the limit of the control variable. This value is compared with the control variable each time the loop is about to be repeated	
increment	is a numeric expression specifying the value to be added (with its algebraic sign) to the control variable when the NEXT statement is encountered	if the STEP option is not specified an increment of +1 is assumed

How FOR/NEXT Statements Work



(*) Unless the initial and final value are equal. In this case the first statement after next is executed

Figure 8-7 FOR/NEXT Statements

Remarks

We shall say that a FOR/NEXT loop is "pending" if it has not yet been completed when a break is encountered. Any modification to the resident program (deleting, editing lines, and so on) will prohibit the loop from resuming execution.

Value of Increment Positive

IF...	THEN...
the value of increment is positive	the FOR/NEXT loop is executed until the value of the control variable is not greater than the final value.

For example:

```
LIST
1Ø K =1Ø
2Ø FOR I=1 TO 1Ø STEP 2
3Ø PRINT I;
4Ø K=K+1Ø
5Ø PRINT K
6Ø NEXT
Ok
RUN

 1  2Ø
 3  3Ø
 5  4Ø
 7  5Ø
 9  6Ø
Ok
```

Here the loop executes five times

the value of increment is positive AND IF the initial value exceeds the final value	the loop does not execute.

For example:

```
LIST
1Ø J=Ø
2Ø FOR I=1 TO J
3Ø PRINT I
4Ø NEXT I
5Ø PRINT "Exit of the loop"
Ok
RUN
Exit of the loop
Ok
```

Value of Increment Negative

IF...	THEN...
the value of increment is negative	the FOR/NEXT loop is executed until the value of the control variable is less than the final value.

For example:

```
LIST
1Ø FOR I%=1 TO -1Ø STEP -3
2Ø PRINT I%;
3Ø NEXT I%
4Ø PRINT
5Ø PRINT "Exit ";
   "CONTROL VARIABLE=";I%
Ok
RUN

 1 -2 -5 -8
Exit CONTROL VARIABLE=-11
```

Here the loop executes four times. When it is exited, control variable maintains its last value (-11), which is displayed by statement 5Ø

the value of increment is negative AND IF initial value is less than final value	the loop does not execute. For example:

```
LIST
1Ø FOR K%=1 TO 1Ø STEP -2
2Ø PRINT K%;
3Ø NEXT K%
4Ø PRINT "Exit ";
   "CONTROL VARIABLE=";K%
Ok
RUN
Exit CONTROL VARIABLE=1
Ok
```

Value of Increment Zero

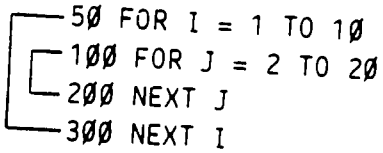
IF...	THEN...
the value of the increment is zero	the loop is executed indefinitely (unless the initial and final values are equal; in this case the loop will not be executed at all). For example:

```
LIST
1ØØ FOR A%=1 TO 3Ø STEP Ø
11Ø PRINT A%;
12Ø NEXT A%
Ok
RUN
 1  1  1...
```

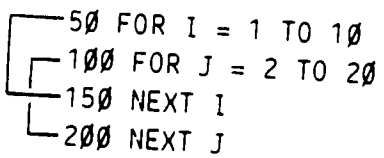
You have to press CTRL C to interrupt execution

Nested Loops

FOR/NEXT loops may be nested one within the other as long as the internal FOR/NEXT loop is entirely within the outer FOR/NEXT loop. For example, the following nesting is valid:



while the following is not:



Nested FOR/NEXT loops cannot use the same control variable.

Each FOR statement specified must have a corresponding NEXT statement.

If nested loops have the same end point, a single NEXT statement may be used for all of them (with a list of control variables).

When a nested loop is encountered it is executed, when it is exited the first statement following the associated NEXT statement will be executed.

Loops may be nested to any depth.

The number of simultaneously active loops is only limited by the amount of memory available.

For example:

```
50 FOR I = 1 TO 10
 .
 .
 .
100 FOR J = 2 TO 20
 .
 .
 .
200 NEXT J,I
```

Nested loops provide a very useful programming technique for solving a wide range of problems. An example of a nested loop is shown below.

Example

DISPLAY

```
LIST
1Ø REM PRIME NUMBERS
2Ø INPUT "Enter limits N,M";N,M
3Ø PRINT "Primes from";N;"TO";M
4Ø PRINT
5Ø PRINT
6Ø FOR I=N TO M
7Ø LET K=SQR(I)
8Ø FOR J=2 TO K
9Ø LET E=I/J-INT(I/J)
1ØØ IF E=Ø THEN 13Ø
11Ø NEXT J
12Ø PRINT I;
13Ø NEXT I
14Ø PRINT
15Ø PRINT
16Ø PRINT "End of List"
17Ø END
Ok
RUN
Enter limits N,M? 1,15
Primes from 1 TO 15


 1  2  3  5  7  11  13


End of List
Ok
```

COMMENTS

you will display all the prime numbers within a given range of numbers. One FOR/NEXT loop specifies the range of the numbers to be used. Nested within that loop is a second loop, one that contains an algorithm to determine if any number in the specified range is a prime number.

To explain the algorithm: numbers assigned to a variable (I) are divided by an integer (J) whose value ranges from 2 to the square root of I. If the remainder of the division is Ø then I is not a prime number, so the number I+1 is generated and the process repeated. The choice of the final value square root is made because if there are any integer factors of the number I they will always lie between 2 and the square root of I

Note: Statement 1ØØ allows you to exit the inner loop even if J is not greater than K. You can always exit a loop by an IF...THEN or GOTO statement, however you cannot enter the loop in any statement other than the initial FOR

Remarks

- if a NEXT statement is encountered before its corresponding FOR statement, a

  NEXT without FOR

  error message is issued and execution is terminated.

For example:

```
1200 IF A>5 THEN 2010

      :
      :
2000 FOR J=1 TO 7
2010 PRINT "HELLO";
2020 NEXT J
```

When executing statement 2020 following a jump from 1200, BASIC displays the above mentioned error message and enters Command Mode

- the final value is always set before the initial value is set.

  For example, if you write:

```
10 I=5
20 FOR I=1 TO I+5
```

  statement 20 will assign the value 10 to the final value. However, for program readability, we do not advise you to use the control variable to define the final value

- if possible use an integer variable for the control variable and integer constants (or integer variables) for the initial and final value and the increment. This will improve the speed of execution.

WHILE/WEND (PROGRAM/IMMEDIATE)

Executes a series of statements in a loop as long as a given condition is true.

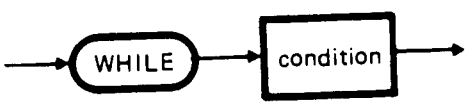


Figure 8-8 WHILE Statement

Figure 8-9 WEND Statement

Where

SYNTAX ELEMENT	MEANING
condition	may be: - a numeric expression - a relational expression - a logical expression Note: BASIC determines whether the condition is true or false by testing the result of the expression for non zero and zero respectively. A non zero result is true and a zero result is false. Because of this, you can test whether the value of a variable is non zero or zero by merely specifying the name of the variable as a condition

How WHILE/WEND Statements Work

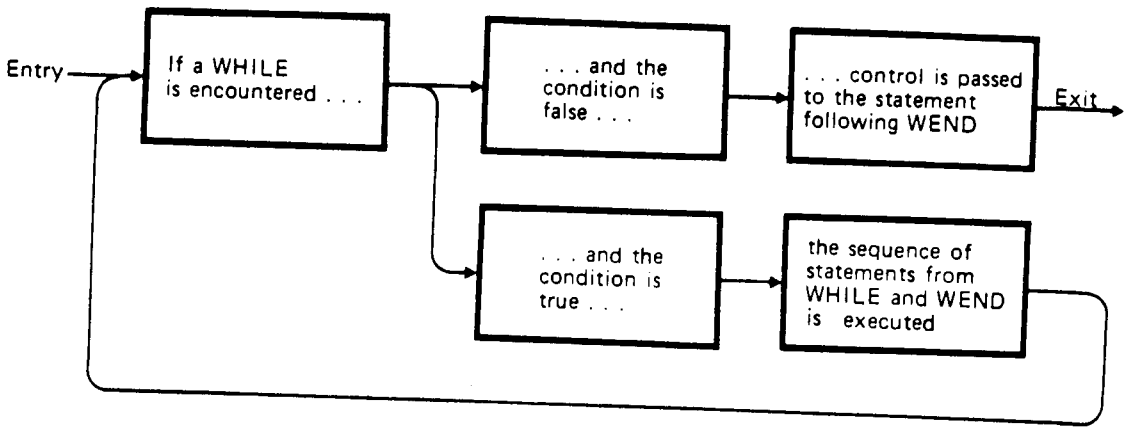


Figure 8-1Ø WHILE/WEND Statements

Remarks

We shall say that a WHILE/WEND loop is "pending" if it has not yet been completed when a break is encountered. Any modification to the resident program (deleting or editing lines, and so on...) will prohibit the loop from resuming execution.

Example

DISPLAY

```
LIST
9Ø 'BUBBLE SORT ARRAY A$
1ØØ FLIPS=1 'FORCE ONE PASS THRU LOOP
11Ø WHILE FLIPS
115 FLIPS=Ø
12Ø FOR I=1 TO J-1
13Ø IF A$(I)>A$(I+1) THEN
    SWAP A$(I),A$(I+1):FLIPS=1
14Ø NEXT I
15Ø WEND
Ok
RUN
Ok
```

COMMENT

you sort the elements of array A$ in ascending value order (from subscript 1 to subscript J)

Note: the condition (in this case the value of variable FLIPS) may be changed during the loop (see line 13Ø)

Remarks

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

You can exit a WHILE/WEND loop either when the condition after WHILE is false or by an IF...THEN or GOTO statement, but you cannot enter the loop in any statement other than the initial WHILE.

9. FUNCTIONS

ABOUT THIS CHAPTER

This chapter describes the intrinsic (built-in) functions, which may be called by any program without further definition and user-defined functions which once set up can be used in exactly the same way but only within the program containing the definition.

CONTENTS

FUNCTIONS

INTRODUCTION

There are occasions when identical expressions are required a number of times in the same program.

To avoid writing these expressions more than once and to save storage, functions can be written and then activated from many places in a BASIC program.

Each function can be called simply by stating its name followed, in parentheses, by one or more "arguments" representing the values the function parameters are to assume. Each argument is associated with a parameter in the function definition.

Arguments are separated by commas. An argument may be a constant, a variable, or an expression.

Parameters are separated by commas too. A parameter may only be a variable.

The number of arguments must be the same as the number of parameters in the function definition and their types (numeric or string) must match. The association between arguments and parameters is positional, i.e. the first argument will be associated with the first parameter etc.... We can pass one or several arguments to a function, or no argument at all.

Numeric conversions are valid from one numeric argument to the corresponding parameter, if it is a different numeric type. If for instance, a floating point value is supplied where an integer is required, BASIC will round the fractional portion and use the resulting integer.

A function returns a single value, which may be a numeric or a string value, depending on the type of the expression used to define the function.

We can classify BASIC functions into two main categories:

- Intrinsic (or built-in) functions

  Built-in functions are an intrinsic part of BASIC. They provide a set of commonly used numeric and string operations. The user can invoke them without an explicit definition within any program. A complete list and a detailed description of built-in functions will be given below.

- User defined functions

  The user can define an arbitrary number of functions in a BASIC program, by the statement DEF FN. The name of a user defined function begins with FN and may be any valid variable name.

Each function definition must precede any function call in the program.

Examples

DISPLAY	COMMENTS
1Ø A=X*SIN(X)+LOG(X)	here SIN and LOG are built-in numeric functions
LIST 1Ø DEF FNH(X,Y)=SQR(X*X+Y*Y) 2Ø INPUT "SIDES";X1,Y1 3Ø PRINT "H=";FNH(X1,Y1); " X1=";X1;" Y1=";Y1 4Ø GOTO 2Ø Ok RUN SIDES? 3.5,1.2 H= 3.7 X1= 3.5 Y1= 1.2 SIDES? 1.7,4 H= 4.34626 X1= 1.7 Y1= 4 SIDES? ^C Break in 2Ø Ok	FNH is a user-defined function. A DEF FN statement defines it (see statement 1Ø). It calculates the square root of the sum of the squares of the parameters X and Y (by using the built-in function SQR). Statement 3Ø calls the user-defined function and passes two arguments to the corresponding parameters. Note: The names of the arguments need not be the same as the names of the corresponding parameters

Remarks

Functions may be used in both immediate and program lines. This information is not specified each time a function is introduced.

USER DEFINED FUNCTIONS

If a numeric or string equation is to be used several times, it is more convenient to define the equation as a function. Once defined, the function can be called in exactly the same way as a built-in function. The only limitation is that the definition is program dependent and must therefore be redefined in each program that needs to use it (unless the second program is CHAINed to the first, with the MERGE option).

DEF FN (PROGRAM)

DEF FN defines a numeric or string function.

A DEF FN statement must be executed before the function it defines can be called.

A DEF FN statment is not permitted in immediate mode.

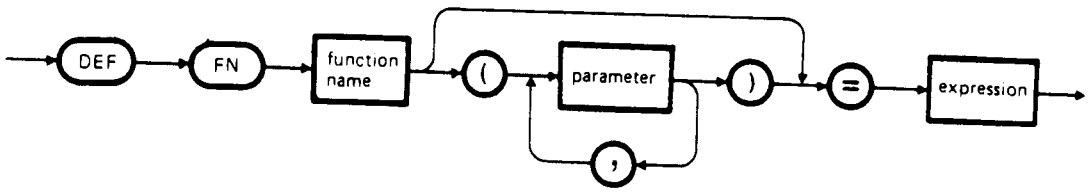


Figure 9-1 DEF FN Statement

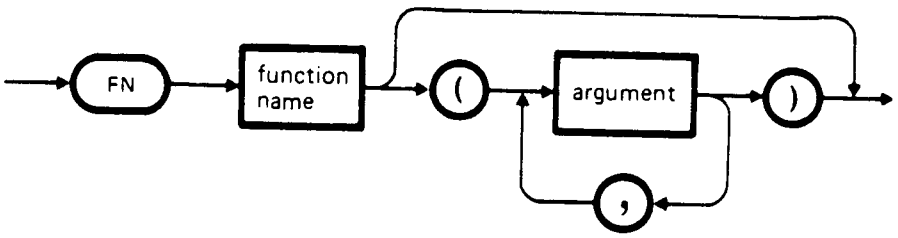


Figure 9-2 Function Call

Where

SYNTAX ELEMENT	MEANING
function name	a legal variable name beginning with FN (numeric or string names may be specified). No blanks may be inserted between FN and the remainder of the name and the first character after FN must be a letter. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement

parameter	a "dummy" variable that is to be replaced by the corresponding argument value when the function is called. The association between arguments and parameters is positional (i.e. the first argument is associated to the first parameter etc.)
argument	the actual value to be passed to the corresponding parameter. Each argument may be a constant, a variable, or an expression
expression	an expression that performs the operation of the function. The type of expression must agree with the type (numeric or string) of the function. The expression normally includes only parameters as variables, but it may also include program variables defined outside the function definition (global variables). Parameter names that appear in the expression serve only to define the function, they do not affect program variables that have the same name. However, for program readability, we do not advise you to use the same names

Characteristics

IF...	THEN...
an argument type does not agree with the corresponding parameter type	a "Type mismatch" error occurs
a user-defined function is called before it has been defined	an "Undefined user function" error occurs

a user-defined function is called by another user-defined function	the called function must be defined in the same program and preceed the call. For example: 10 DEF FNA(X)=(SIN(X/5)*3.1)/180 20 DEF FNB(X)=(FNA(X)+SIN(X))*.5
a program CHAINs another program with the option MERGE	function definitions must be placed in the CHAINing program before the CHAIN statement. Otherwise, the user-defined functions will be undefined when the merge is complete. (For more details see Chapter 11). For example: 10 DEF FNA(X)=(X+X*(X+1)) : 100 CHAIN MERGE "V1:PROG1"

Remark

The syntax of the Function Call is valid both for user-defined and built-in functions.

BUILT-IN NUMERIC FUNCTIONS

BASIC provides a number of pre-written routines, that save you the effort of writing groups of statements to calculate such mathematical functions as square root, sine and natural logarithm. With the exception of CDBL, which returns a double precision result, only integer and single precision results are returned by built-in numeric functions.

All the built-in numeric functions are listed in alphabetical order, below.

Note: In this list we also include the RANDOMIZE statement, as it is closely related to the RND function.

ABS

Returns the absolute value of a numeric expression.

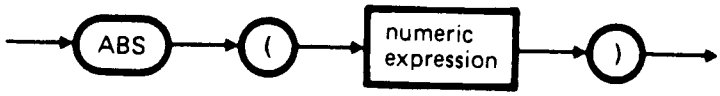


Figure 9-3 ABS Function

Example

```
PRINT ABS(7*(-5))
 35
Ok
```

ATN

Returns the arctangent of the argument.

The value returned is expressed in radians and falls in the range $-\pi/2$ to $\pi/2$ (where π is 3.1415...).

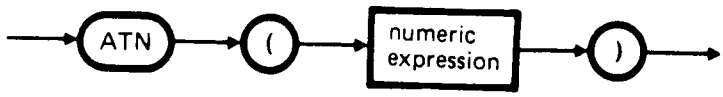


Figure 9-4 ATN Function

Example

```
1Ø INPUT X
2Ø PRINT ATN(X)
Ok
RUN
? 3
 1.249Ø5
Ok
```

Remark

The evaluation of ATN is performed in single precision.

CDBL

CDBL converts any numeric type to a double precision (8 bytes) argument.

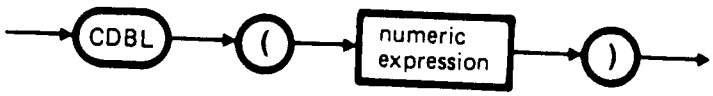


Figure 9-5 CDBL Function

Example

```
1Ø A = 454.67
2Ø PRINT A;CDBL(A)
RUN
 454.67  454.67ØØ13427734
Ok
```

CINT

Converts any numeric type argument to an integer by rounding the fractional part (if the fraction is > = .5 the integer part is rounded up, otherwise a truncation occurs).

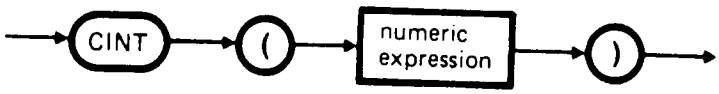


Figure 9-6 CINT Function

Example

```
PRINT CINT(45.67)
 46
Ok
```

Remarks

If the argument is a value outside the range -32768 and 32767, an "Overflow" error occurs.

See also FIX and INT, which also return integer values.

COS

Returns the cosine of the argument.

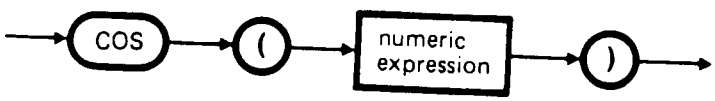


Figure 9-7 COS Function

Example

```
1Ø X = 2*COS(.4)
2Ø PRINT X
RUN
 1.84212
Ok
```

Remarks

The argument passed to the function is assumed to be the value of an angle measured in radians.

The evaluation of COS is performed in single precision.

CSNG

Converts any numeric type argument to a single precision number (4 bytes).

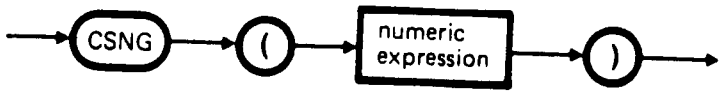


Figure 9-8 CSNG Function

Example

```
1Ø A# = 975.3421#
2Ø PRINT A#; CSNG(A#)
RUN
 975.3421   975.342
OK
```

Remarks

See also CINT and CDBL functions for converting numbers to the integer and double precision data types.

EXP

Raises the constant "e" (e = 2.71828) to the power of the argument.

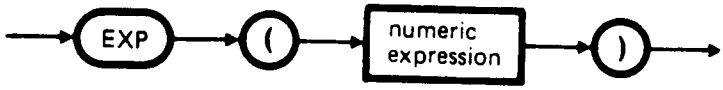


Figure 9-9 EXP Function

Example

```
1Ø X = 5
2Ø PRINT EXP(X-1)
RUN
 54.5981
Ok
```

Remarks

The argument value must be < =88.7228. Otherwise the overflow error message is displayed, machine infinity with the appropriate sign is supplied as the result and execution continues.

The evaluation of EXP is performed in single precision.

FIX

Returns the integer part of the argument (truncation).

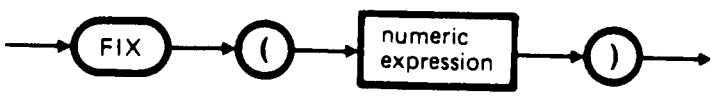


Figure 9-10 FIX Function

Examples

```
PRINT FIX(58.75)
 58
Ok

PRINT FIX(-58.75)
-58
Ok
```

Remarks

FIX(X) is equivalent to SGN(X)*INT(ABS(X)). Unlike INT, FIX does not return the next lower number for negative arguments (see the second example above).

FRE

Returns the number of bytes in memory not being used by BASIC.

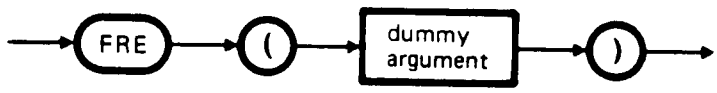


Figure 9-11 FRE Function

Where

SYNTAX ELEMENT	MEANING
dummy argument	is any numeric or string expression. The value returned is not affected by the argument value

Examples

```
PRINT FRE(Ø)
 14542
Ok
```

```
PRINT FRE(X$)
 14542
Ok
```

Remarks

FRE("") forces a garbage collection before returning the number of free bytes. Moreover, BASIC will perform a garbage collection if all memory has been used up.

INT

Returns the largest integer less than or equal to the argument.

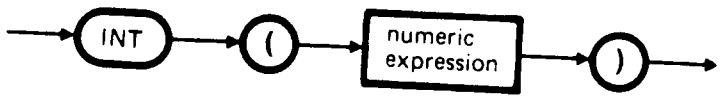


Figure 9-12 INT Function

Examples

```
PRINT INT(99.89)
 99
Ok

PRINT INT(-12.11)
-13
Ok
```

Remarks

Notice the difference between INT and FIX. With negative values, the returned value for INT is always smaller than or equal to the argument, whilst for FIX it is always greater than or equal to the argument.

LOG

Returns the natural logarithm of a positive argument.

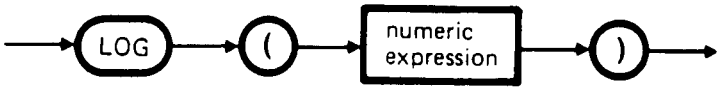


Figure 9-13 LOG Function

Where

SYNTAX ELEMENT	MEANING
numeric expression	must be positive. Otherwise an "Illegal function call" error occurs

Example

```
PRINT LOG(45/7)
 1.86Ø75
Ok
```

Remarks

Since $\log_a x = \frac{\log_e x}{\log_e a}$ the common logarithm (or any other base) can easily be evaluated by use of the LOG function.

If you need this function frequently in a program, it should be specified as a user-defined function.

For example, you may write at the beginning of your program:

```
1Ø DEF FNLOG1Ø(X)=LOG(X)/LOG(1Ø)
```

and call FNLOG1Ø, passing the corresponding argument, anywhere you need.

The evaluation of LOG is performed in single precision.

RND

Returns a random number between Ø and 1. The same sequence of random numbers is generated each time the program is RUN, unless the random number generator is reseeded (see RANDOMIZE statement).

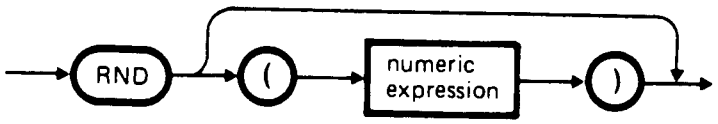


Figure 9-14 RND Function

Where

SYNTAX ELEMENT	MEANING
numeric expression	<Ø restarts the same random number sequence
	=Ø repeats the last number generated
	>Ø (or omitted, i.e. RND) the next random number in the sequence is generated

Example

```
1Ø FOR I=1 TO 5
2Ø PRINT INT(RND*1ØØ);
3Ø NEXT
RUN
 8  25  77  68  7
Ok
```

Remarks

Although it is called Random, the number is actually taken from a fixed cycle of numbers, about one million in all. Since the cycle starts for each run, the same program gives the same result every time it is run. If all the numbers are used, the cycle begins again.

To change the random number sequence every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN (see RANDOMIZE).

RANDOMIZE (PROGRAM/IMMEDIATE)

Reseeds the random number generator.

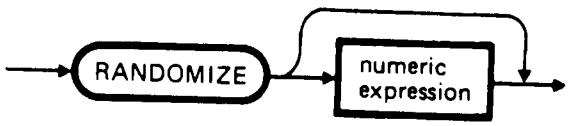


Figure 9-15 RANDOMIZE Statement

Where

SYNTAX ELEMENT	MEANING
numeric expression	must be in the range of integers (-32768 to 32767). If it is not an integer it is rounded to the nearest integer. This number is used to set the starting point (seed) of a new random number sequence. If it is omitted, BASIC suspends program execution and asks for a value by displaying: Random Number Seed (-32768 to 32767)? before executing RANDOMIZE

Remarks

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

You are not limited to random numbers between Ø and 1. To generate the sequence between A and B, use the formula:

```
FIX((B+1-A)*RND+A)
```

Examples

```
1Ø RANDOMIZE
2Ø FOR I=1 TO 5
3Ø PRINT RND;
4Ø NEXT I
RUN
Random Number Seed (-32768 TO 32767)? 3 (user enters 3 CR)
 .88598   .484668   .586328   .119426   .7Ø9225
OK
RUN
Random Number Seed (-32768 to 32767)? 4 (user enters 4 CR)
.8Ø35Ø6   .162462   .929364   .292443   .322921
Ok
RUN
Random Number Seed (-32768 to 32767)? 3 (same sequence as first RUN)
 .88598   .484668   .586328   .119426   .7Ø9225
Ok
```

SGN

Returns 1 if the argument is positive, Ø if the argument is zero and -1 if the argument is negative.

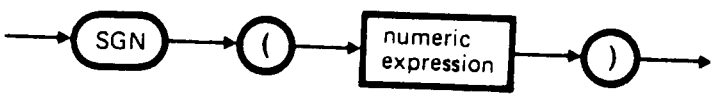


Figure 9-16 SGN Function

Example

```
ON SGN(X)+2 GOTO 1ØØ,2ØØ,3ØØ
```

branches to:

- 1ØØ if X < Ø
- 2ØØ if X = Ø
- 3ØØ if X > Ø

SIN

Returns the sine of the argument.

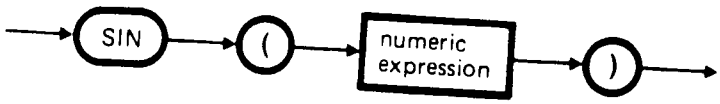


Figure 9-17 SIN Function

Example

```
PRINT SIN(1.5)
 .997495
Ok
```

Remarks

The argument passed to the function is assumed to be the value of an angle measured in radians.

SIN is evaluated as single precision.

SQR

Returns the square root of the argument.

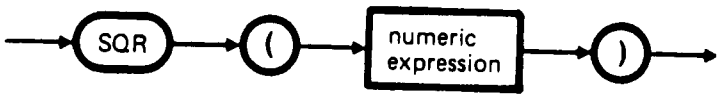


Figure 9-18 SQR Function

Example

```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10            3.16228
 15            3.87298
 20            4.47214
 25            5
Ok
```

Remarks

An "Illegal function call" error results if the argument is negative.

SQR is evaluated in single precision.

TAN

Returns the tangent to the argument.

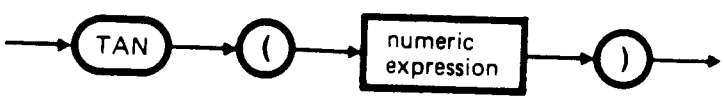


Figure 9-19 TAN Function

Example

```
10 Y = Q*TAN(X)/2
```

Remarks

The value of the argument is assumed to be measured in radians.

If TAN overflows, an "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result and execution continues.

TAN is evaluated in single precision.

BUILT-IN STRING FUNCTIONS

They are intrinsic functions which return a string or numeric value and permit one or more than one numeric and/or string arguments.

They simplify such string operations as extracting group of characters- a substring-from a larger string.

All the built-in string functions are listed in alphabetical order, below.

Note: In this list we also include the MID$ statement, as it is closely related to the MID$ function.

ASC

Returns a numerical value that is the ASCII code of the first character of a given string.

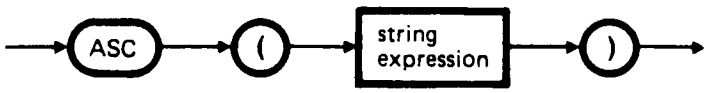


Figure 9-20 ASC Function

Example

```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
```

Remarks

If the string expression argument is the null string (""), an "Illegal function call" error occurs.

See the CHR$ function for ASCII-to-string conversion.

CHR$

Returns a one-character string whose ASCII code is the value of the argument.

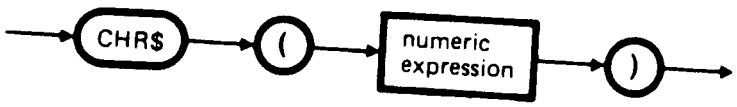


Figure 9-21 CHR$ Function

Where

SYNTAX ELEMENT	MEANING
numeric expression	is evaluated and rounded to the nearest integer. It is interpreted as an ASCII code and must be in the range 0 to 255. Otherwise an "Illegal function call" error occurs

Example

```
PRINT CHR$(66)
B
Ok
```

Remarks

CHR$ is commonly used to send a special character to the terminal. For instance, the character could be sent (CHR$(7)) as a preface to an error message, or a form feed could be sent (CHR$(12)) to clear a CRT screen and return the cursor to the home position.

See the ASC function for ASCII-to-numeric conversion.

HEX$

Converts a decimal number to the corresponding hexadecimal string.

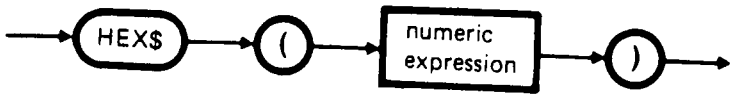


Figure 9-22 HEX$ Function

Where

SYNTAX ELEMENT	MEANING
numeric expression	is rounded to the nearest integer before HEX$ is evaluated

Example

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL"
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
```

Remark

See the OCT$ function for octal conversion.

INKEY$

Returns either a one character string containing a character read from the keyboard or a null string if no character is pending at the keyboard No characters will be echoed and all characters are passed through to the program except for CTRL C which interrupts program execution.

Figure 9-23 INKEY$ Function

Examples

DISPLAY

```
1000 'Timed Input Subroutine
1010 RESPONSE$=""
1020 FOR I%=1 TO TIMELIMIT%
1030 A$=INKEY$:IF LEN(A$)=0 THEN 1060
1040 IF ASC(A$)=13 THEN TIMEOUT%=0:RETURN
1050 RESPONSE$=RESPONSE$+A$
1060 NEXT I%
1070 TIMEOUT%=1:RETURN
```

COMMENTS

This subroutine returns two values:

- RESPONSE$ which contains the string entered from keyboard

- TIMEOUT% which equals 0 if the user enters a string of characters from keyboard before a specified number of loops (TIMELIMIT%) otherwise equals 1

Note: the LEN function is described later in this chapter

INPUT$

Returns a string of a specified length, read from the keyboard or from a disk file. No characters will be echoed and all control characters are passed through except CTRL C which is used to interrupt the execution of the INPUT$ function.

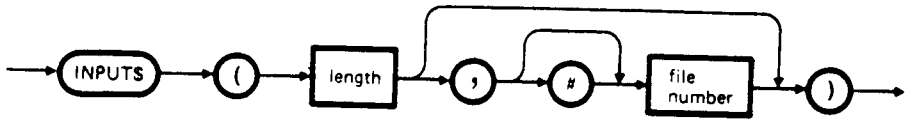


Figure 9-24 INPUT$ Function

Where

SYNTAX ELEMENT	MEANING
length	is a numeric expression rounded to the nearest integer. It specifies the length of the returned string
file number	is the number of the buffer associated with the file (see Chapter 12)

Examples

```
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

this program lists the contents of a sequential file in hexadecimal

Note: EOF equals -1 when the end of file is reached (see Chapter 12)

```
110 X$=INPUT$(1)
120 IF X$="S" THEN END
```

enter S to end the program, or any other character to continue

INSTR

Searches for the first occurrence of a given substring in a given string and returns the position at which the match is found.

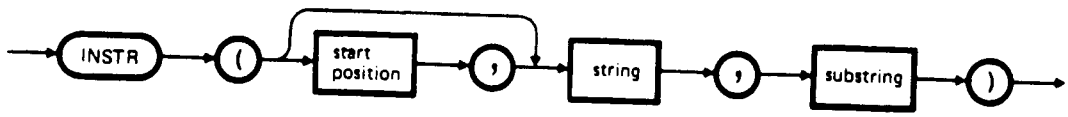


Figure 9-25 INSTR Function

Where

SYNTAX ELEMENT	Meaning
start position	is a numeric expression rounded to the nearest integer which specifies where the search is to begin. Its value must be in the range 1 to 255. If it is omitted 1 is assumed
string	is a string expression whose value is the string to be searched
substring	is either a string constant or string variable whose first occurrence is to be searched for

Example

DISPLAY

```
1Ø X$ = "ABCDEB"
2Ø Y$ = "B"
3Ø PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2  6
Ok
```

COMMENTS

Note that the position at which the match is found is always evaluated from the beginning of the original string, even if a start position is specified

Special Values

IF...	THEN...
start position>LEN(string)	the returned value is Ø
start position falls outside the range 1 to 255	an error message is issued (Illegal function call)
string is empty (null string)	the returned value is Ø
substring cannot be found	the returned value is Ø
substring is empty and start position is specified	the returned value equals the start position value
substring is empty and start position is omitted	the returned value is 1

LEFT$

Returns a substring comprised of the leftmost string characters of a given length.

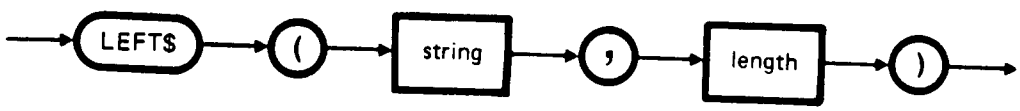


Figure 9-26 LEFT$ Function

Where

SYNTAX ELEMENT	MEANING
string	is a string expression whose value is the string from which the substring is to be returned

length	is a numeric expression rounded to the nearest integer, whose value (from Ø to 255) represents the length of the returned string

Example

```
1Ø A$ = "BASIC LANGUAGE"
2Ø B$ = LEFT$(A$,5)
3Ø PRINT B$
RUN
BASIC
Ok
```

Remarks

IF...	THEN...
length=Ø	the null string is returned
length falls outside the range 1 to 255	an "Illegal function call" error is issued
length > = LEN(string)	the entire string is returned

LEN

Returns the length of a specified string.

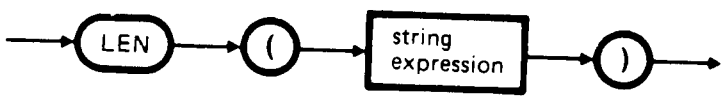


Figure 9-27 LEN Function

Example

```
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
RUN
 16
Ok
```

Remarks

All characters, printable and non printable and blanks are counted by the LEN function.

MID$

Returns a substring from a specified string, starting from a specified character position. The length of the returned substring can be specified, or all the characters to the end of the string are returned.

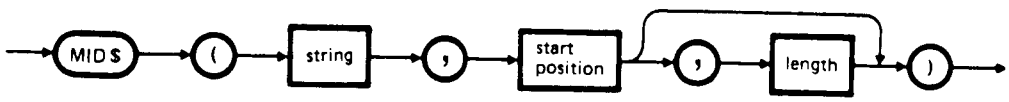


Figure 9-28 MID$ Function

Where

SYNTAX ELEMENT	MEANING
string	is a string expression whose value is the string from which the substring is to be returned
start position	is a numeric expression rounded to the nearest integer, whose value (>=1 and <= the length of string) specifies the character position of the beginning of the returned substring

length	is a numeric expression rounded to the nearest integer, whose value (from Ø to 255) represents the length of the returned substring. If length is omitted all the characters from start position to the end of the string are returned. If length = Ø the null string is returned

Example

```
LIST
1Ø A$="GOOD "
2Ø B$="MORNING EVENING AFTERNOON"
3Ø PRINT A$;MID$(B$,9,7)
Ok
RUN
GOOD EVENING
Ok
```

Remarks

IF...	THEN...
start position>LEN(string)	MID$ returns a null string
start position=Ø	the error message "Illegal function call" will be displayed
length is omitted OR there are fewer characters left than length specifies	all the characters from start position to the end of the string are returned

MID$ (PROGRAM/IMMEDIATE)

Replaces a portion of a string with another string

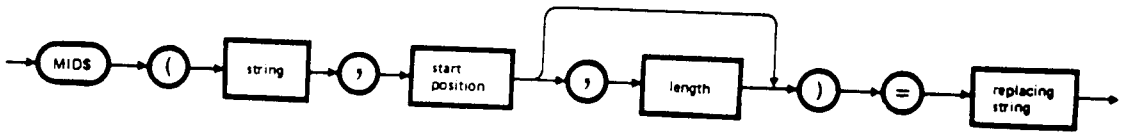


Figure 9-29 MID$ Statement

Where

SYNTAX ELEMENT	MEANING
string	is a string variable whose value is the string from which a substring is to be replaced
start position	is a numeric expression rounded to the nearest integer, whose value (>= 1 and <= the length of string) specifies the character position where the replacement has to begin.
length	is a numeric expression rounded to the nearest integer, whose value (from Ø to 255) represents the length of the returned string. If length is omitted all the characters from start position to the end of the replacing string are replaced. However, regardless of whether length is omitted or included, the replacement of characters never goes beyond the original lenght of string. If length = Ø the null string is returned.
replacing string	is a string expression which replaces the characters in the original string, beginning at start position.

Example

```
1Ø A$="KANSAS CITY, MO"
2Ø MID$(A$,14)="KS"
3Ø PRINT A$
RUN
KANSAS CITY, KS
```

Remarks

IF...	THEN...
start position>LEN(string)	MID$ returns a null string
start position=Ø	the error message "Illegal function call" will be displayed
length is omitted	all the characters from start position to the end of the replacing string will be replaced
length=Ø	the null string is returned
an attempt is made to replace characters beyond the original length of the string	the replacement of characters ends at the last character of the original string

OCT$

Returns string which represents the octal value of a decimal argument.

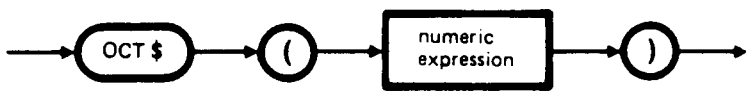


Figure 9-30 OCT$ Function

Where

SYNTAX ELEMENT	MEANING
numeric expression	is rounded to the nearest integer before OCT$ is evaluated

Example

```
PRINT OCT$(24)
 3Ø
Ok
```

Remark

See the HEX$ function for hexadecimal conversion.

RIGHT$

Returns a substring from a specified string, extracting its rightmost characters.

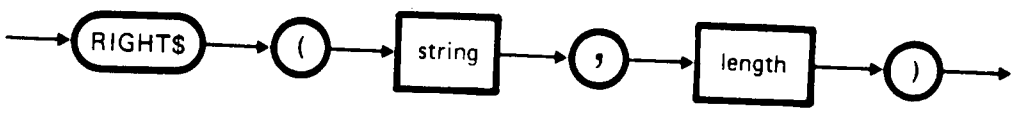


Figure 9-31 RIGHT$ Function

Where

SYNTAX ELEMENT	MEANING
string	is a string expression whose value is the original string from which a substring is to be returned
length	is a numeric expression rounded to the nearest integer, whose value (from Ø to 255) represents the length of the returned substring

Example

```
1Ø A$="DISK BASIC"
2Ø PRINT RIGHT$(A$,5)
RUN
BASIC
Ok
```

Remarks

IF...	THEN...
length=Ø	the null string (length zero) is returned
length>=LEN(string)	the entire original string is returned

SPACE$

Returns a string of a specified number of spaces.

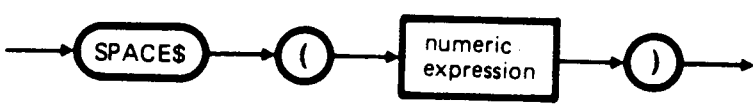


Figure 9-32 SPACE$ Function

Where

SYNTAX ELEMENT	MEANING
numeric expression	is rounded to the nearest integer and must be in the range Ø to 255 (to avoid "Illegal function call" error). It specifies the number of spaces, i.e. the length of the returned string

Example

```
1Ø FOR I=1 TO 5
2Ø X$=SPACE$(I)
3Ø PRINT X$;I
4Ø NEXT I
RUN
  1
   2
    3
     4
      5
Ok
```

Remark

Also see the SPC function in the next paragraph.

STR$

Converts a numeric expression to a string.

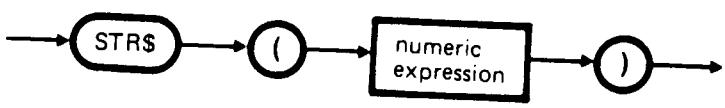


Figure 9-33 STR$ Function

Examples

DISPLAY	COMMENTS

`5 REM ARITHMETIC FOR KIDS`
`1Ø INPUT "TYPE A NUMBER";N`
`2Ø ON LEN (STR$(N)) GOSUB 3Ø,1ØØ,`
`2ØØ,3ØØ,4ØØ,5ØØ`
`:`
`:`
	The entered number N is converted to a string by the STR$ function

`LIST`
`1Ø A$=STR$(7Ø)`
`2Ø PRINT A$`
`Ok`
`RUN`
`7Ø`
`Ok`
	7Ø (the argument of STR$ is a number, but the contents of A$ is a two character string whose value is 7Ø)

`LIST`
`1Ø A!=1.3`
`2Ø A#=VAL(STR$(A!))`
`3Ø PRINT A#`
`Ok`
`RUN`
`1.3`
`Ok`
	The conversion in line 2Ø causes the value in A! to be stored accurately in double-precision variable A#

Remark

VAL performs the opposite function (see VAL).

STRING$

Returns a string of specified length, whose characters are all the same specified ASCII code value, or are all the first character of a specified string.

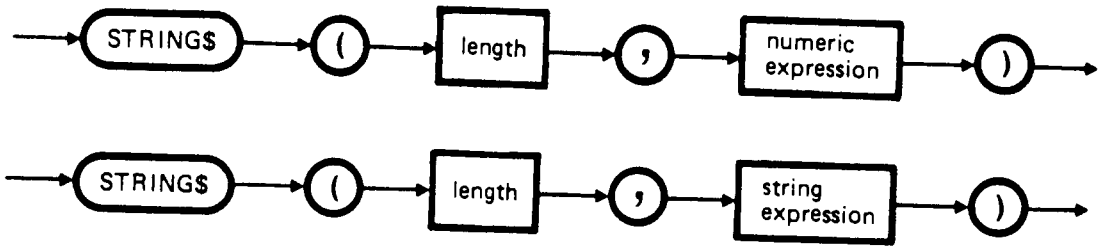


Figure 9-34 STRING$ Function

Where

SYNTAX ELEMENT	MEANING
length	is a numeric expression rounded to the nearest integer. It specifies the length (from Ø to 255) of the resulting string
numeric expression	is rounded to the nearest integer. It specifies the ASCII decimal code (from Ø to 255) whose corresponding character is used to form the resulting string
string expression	is evaluated. Its first character is used to form the resulting string

Example

```
1Ø X$=STRING$(1Ø,45)
2Ø PRINT X$"MONTHLY REPORT"X$
RUN
----------MONTHLY REPORT----------
Ok
```

VAL

Converts the string representation of a number to its numeric value.

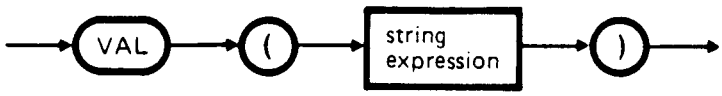


Figure 9-35 VAL Function

Where

SYNTAX ELEMENT	MEANING
string expression	is evaluated. Leading and trailing blanks, tabs, and linefeeds (if any) are stripped away. The remaining string is converted to a number (if it is a valid numeric representation value, otherwise VAL returns Ø). For example: VAL(" -3") is 3 VAL("ABC") is Ø

Example

```
1Ø READ NAME$,CITY$,STATE$,ZIP$
2Ø IF VAL(ZIP$)<9ØØØØ OR VAL(ZIP$)>96699 THEN
   PRINT NAME$ TAB(25) "OUT OF STATE"
3Ø IF VAL(ZIP$)>=9Ø8Ø1 AND VAL(ZIP$)<=9Ø815 THEN
   PRINT NAME$ TAB(25) "LONG BEACH"
```

Remark

The STR$ function performs the opposite task (see STR$).

INPUT/OUTPUT AND SPECIAL BUILT-IN FUNCTIONS

These functions perform the various tasks to do with input/output, value conversions, error handling, carriage positions, memory locations, etc.

They are listed in alphabetic order below.

Note: This section also includes the reserved string words DATE$ and TIME$ (which may be used as functions or as variables depending on whether they appear in an expression or to the left side of the equal sign in an assignment statement).

DATE$/TIME$

Are PCOS elements that are readable or changeable in BASIC by referencing these reserved strings.

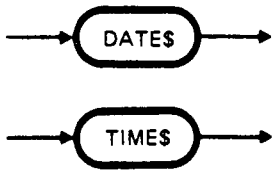


Figure 9-36 DATE$ and TIME$

Remarks

Date and Time may be set either in PCOS by the SSYS command or in BASIC by an assignment statement. The date is entered either as mm:dd:yy, or mm:dd:yyyy. The time is entered as hh:mm:ss. The user can use his own delimiter. (Any printable ASCII character, excluding digits).

For more details see "Professional Computer Operating System (PCOS) User Guide".

Example

DISPLAY	COMMENTS
1ØØ IF DATE$="Ø4:3Ø:82" THEN 3ØØØ	statement 1ØØ checks the date.
. . .	Statement 5ØØ sets the date, and also changes the delimiter.
5ØØ DATE$="Ø5/Ø6/1981"	
. . .	Statement 6ØØ displays the time.
6ØØ PRINT TIME$	Statement 7ØØ set the time
. . .	
7ØØ TIME$="Ø7:4Ø:15"	

CVD

Converts an 8-character string to a double precision number.

See Chapter 12.

CVI

Converts a 2-character string to an integer.

See Chapter 12.

CVS

Converts a 4-characters string to a single precision number.

See Chapter 12.

EOF

Returns true(-1) if the end of a sequential file has been reached.

See Chapter 12.

ERL

Returns the line number of the line in which an error was detected.

See Chapter 13.

ERR

Returns the error code number.

See Chapter 13.

LOC

Returns the record number just read or written (random files), or the number of sectors read or written since the file was OPENed (sequential files).

See Chapter 12.

LPOS

Returns the current position of the connected line printer print head within the line printer buffer.

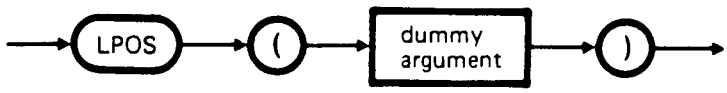


Figure 9-37 LPOS Function

Where

SYNTAX ELEMENT	MEANING
dummy argument	is any numeric or string expression. The returned value is not affected by the value of the argument

Example

```
100 IF LPOS(X)>30 THEN LPRINT CHR$(13)
```

MKD$

Converts a double precision value to an 8-character string.

See Chapter 12.

MKI$

Converts an integer to a 2-character string.

See Chapter 12.

MKS$

Converts a single precision value to a 4-character string.

See Chapter 12.

SPC

Inserts spaces in PRINT or LPRINT statements.

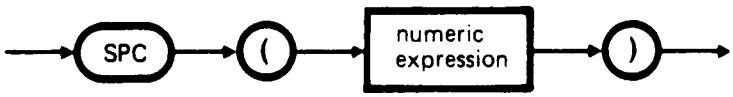


Figure 9-38 SPC Function

Where

SYNTAX ELEMENT	MEANING
numeric expression	is rounded to the nearest integer. It specifies the number of spaces to be inserted in the output image either between two output items or at the beginning or the end of the image. It must be in the range Ø to 255 (to avoid an "Illegal function call" error)

Example

```
PRINT "OVER" SPC(15) "THERE"
OVER               THERE
Ok
```

Remarks

Either a semicolon (;) or a blank follows SPC in a PRINT or LPRINT statement.

See also the SPACE$ function.

TAB

Tabs the cursor or the print head to a specified position, in PRINT or LPRINT statements.

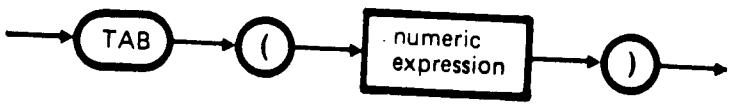


Figure 9-39 TAB Function

Where

SYNTAX ELEMENT	MEANING
numeric expression	is rounded to the nearest integer. The expression must be in the range 1 to 255 (to avoid "Illegal function call" error). 1 is the left hand limit, width minus one is the righthand limit. It specifies the precise cursor (or print head) position in a line

Examples

```
10 PRINT "NAME" TAB(25) "AMOUNT":PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G.T.JONES","$25.00"
RUN
NAME                    AMOUNT

G.T.JONES               $25.00
Ok
```

Remark

If the current cursor or print head position is beyond the value of the argument, TAB goes to that position on the next line.

VARPTR

Format 1 (below). Returns the address in memory of the first byte of data associated with the specified variable.

Format 2. For sequential files, returns the starting address of the disk I/O buffer associated with the file. For random files, returns the address of the FIELD buffer associated with the file.

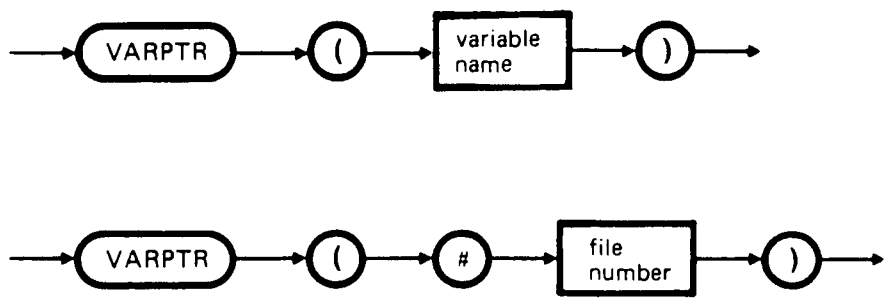


Figure 9-4 VARPTR Function

Where

SYNTAX ELEMENT	MEANING
variable name	any type of variable (numeric string or array). The address returned will be an integer in the range -32768 to 32767. Note: If a negative address is returned, add 65536 to obtain the actual address
file number	the number of the buffer associated with the file.

Example

```
10Ø X%=VARPTR(A(Ø))
```

Remarks

A value must be assigned to the variable prior to execution of VARPTR, if it is a simple variable. Otherwise an "Illegal function call" error results.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subprogram. A function call of the form VARPTR(A(Ø)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

10. SUBPROGRAMS

ABOUT THIS CHAPTER

Often, the same sequence of statements must be executed more than once within a program. In this case you need not reproduce that sequence several times. You may parcel it up as a subprogram and simply call that subprogram from various places in your program. At the end of each execution of the subprogram control goes back to the statement following the call.

The M20 provides you with two kinds of subprogram which may be called by a BASIC program:

- subprograms written in BASIC (we shall call them "BASIC Subroutines")
- subprograms written in the M20 ASSEMBLER i.e., PCOS commands or other assembler subprograms.

This chapter will illustrate these two kinds of subprograms and how to call them when you are in BASIC.

CONTENTS

SUBPROGRAMS

BASIC SUBROUTINES

A BASIC subroutine is formed by a sequence of BASIC statements and it is an integral part of the program. Usually (but not necessarily) a BASIC subroutine begins with a REM statement and ends with a RETURN statement. It is good programming practice always to insert subroutines one after the other at the end of the program and write an END, GOTO, or STOP statement before the first statement of the first subroutine (to avoid "falling" into the subroutine block).

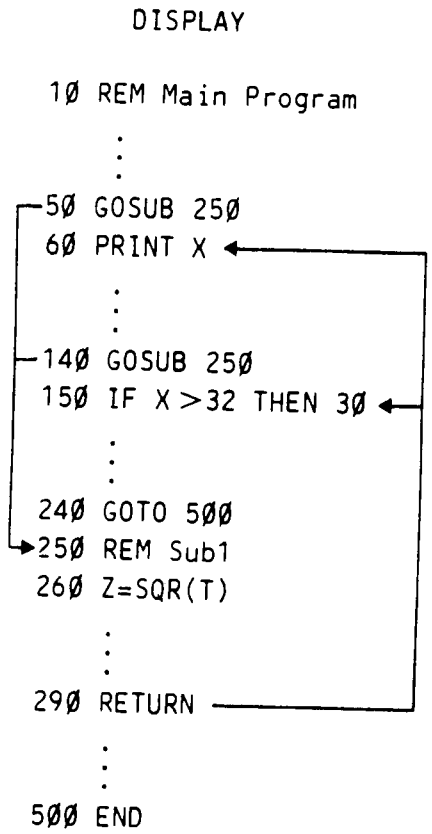
A subroutine is called by a GOSUB or an ON...GOSUB statement. At the end of the execution of a subroutine, control is returned to the first statement following the most recent GOSUB (or ON...GOSUB) that has been executed.

We shall call a BASIC subroutine "pending" if it has not yet been completed when a break is encountered. Any modification to the resident program (deleting, or editing lines, and so on), will prevent the subroutine from resuming execution.

The following example illustrates the call mechanism (statements GOSUB and RETURN).

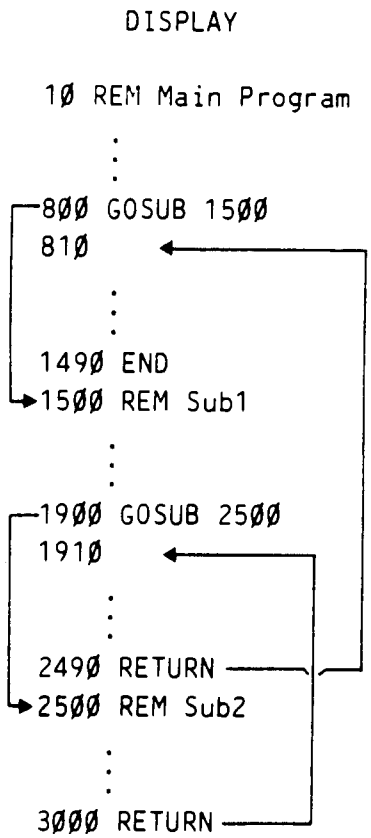
DISPLAY	COMMENTS
1Ø REM Main Program : 5Ø GOSUB 25Ø 6Ø PRINT X : 24Ø GOTO 5ØØ 25Ø REM Sub1 26Ø Z=SQR(T) : 29Ø RETURN : 5ØØ END	when statement 5Ø is encountered (GOSUB), control is passed to statement number 25Ø (which is the first statement of the subroutine). The subroutine is then executed and when statement 29Ø (the RETURN statement) is encountered, control is transferred back to statement 6Ø, the first statement after GOSUB. The statement 24Ø (GOTO) prevents falling into the subroutine by directing control of execution around it

If a program refers to the same subroutine more than once, control is always returned from the subroutine to the statement following the most recent GOSUB (or ON...GOSUB) executed. For example, consider a program that contains the following statements:



A subroutine may also be called by another subroutine. In this case we say that the called subroutine is "nested" within the calling one. The process may be repeated to any depth; the number of nested active subroutines is only limited by the amount of memory available. (An active subroutine is a subroutine where RETURN has not yet been executed). Each GOSUB, whether in the main program or in a subroutine, is always associated with a RETURN statement.

This RETURN statement causes control to be transferred to the first statement following GOSUB. This kind of association is made dynamically (i.e., at run-time), the first RETURN executed is associated with the most recent GOSUB executed, the second RETURN with the next most recent GOSUB and so on.



GOSUB/RETURN (PROGRAM)

GOSUB calls a BASIC subroutine by branching to the specified line number.

RETURN transfers control to the statement following the most recent GOSUB (or ON...GOSUB) executed.

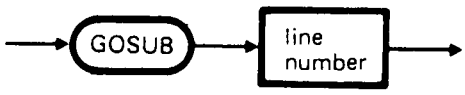


Figure 10-1 GOSUB Statement

Figure 10-2 RETURN Statement

Where

SYNTAX ELEMENT	MEANING
line number	is the first line of a BASIC subroutine

Characteristics

A SUBROUTINE MAY...	COMMENTS
begin with any statement other than NEXT or WEND	for example a subroutine might begin with REM, LET, FOR, ... etc. It is good programming practice to begin a subroutine with a REM statement (or a statement with a comment field)
finish with a RETURN statement	it is good programming practice to finish a subroutine with a RETURN statement. In any case RETURN must be the last statement executed in a subroutine, as RETURN is the only statement that allows control to be returned to the main program. A subroutine may also contain more than one RETURN statement (for instance, if a subroutine has several branches, any of which require a return to be made to the main program)

be called anywhere and any number of times in a program	if a program calls the same subroutine more than once, control is returned, after execution of the subroutine, to the statement following the GOSUB (or ON...GOSUB) that was last executed
be placed anywhere in the program	however it is good programming practice to write subroutines one after the other at the end of the program. To avoid "falling" into a subroutine write an END, or GOTO or STOP statement before the first statement of a subroutine
call another subroutine	the number of nested active subroutines is only limited by the amount of memory available
access any program variable	all variables defined in the "main" program ("global variables") are available to the subroutines. Therefore subroutines may work on program variables without restriction (even modifying their values if need be)

Examples

DISPLAY

```
LIST
1Ø DEFINT A-Z 'defines all integers
2Ø INPUT "Enter 3 integers";A,B,C
3Ø LET X=A
4Ø LET Y=B
5Ø GOSUB 11Ø
6Ø LET X=G
7Ø LET Y=C
8Ø GOSUB 11Ø
9Ø PRINT "The GCD of";A;B;C;"is";G
1ØØ GOTO 19Ø
11Ø LET Q=INT (X/Y) 'routine GCD
```

COMMENTS

on the left is a complete program illustrating a subroutine. The subroutine uses Euclid's Algorithm to find the greatest common divisor (GCD) of three integers. The user enters three integer numbers from the keyboard. The first two numbers entered (A and B) are assigned to X and Y respectively (see statements 3Ø and 4Ø) and their GCD is determined in the subroutine (statements 11Ø to 18Ø). The GCD just found is

```
12Ø LET R=X-Q*Y
13Ø IF R=Ø THEN 17Ø
14Ø LET X=Y
15Ø LET Y=R
16Ø GOTO 11Ø
17Ø LET G=Y
18Ø RETURN
19Ø END
Ok
RUN
Enter 3 integers? 1377,2916,4Ø5
The GCD of 1377  2916  4Ø5 is 81
Ok
RUN
Enter 3 integers? 4,3333,67
The GCD of 4  3333  67 is 1
Ok
```

assigned to X in statement 6Ø and the third number (C) is assigned to Y in statement 7Ø. The subroutine is called again from statement 8Ø to find the GCD of these two numbers. This result is the GCD of the three integers entered. These three numbers, with their GCD, are displayed by statement 9Ø.

Note: Statement 1Ø defines all variables as integer variables as the program works on integer numbers only

```
LIST
1Ø INPUT "Enter N>Ø";N%
2Ø IF N%<=Ø THEN 1Ø
3Ø GOSUB 5Ø
4Ø END
5Ø REM SUB1(Sum of Integers)
6Ø S%=(N%*(N%+1))/2
7Ø PRINT "Sum of Integers
   from 1 to ";N%;"=";S%
8Ø INPUT "Sum of Squares (Y/N)";X$
9Ø IF X$="Y" THEN GOSUB 11Ø
1ØØ RETURN
11Ø REM SUB2(Sum of Squares)
12Ø S2%=(N%*(N%+1)*(2*N%+1))/6
13Ø PRINT "Sum of Squares
   from 1 to ";N%;"=";S2%
14Ø RETURN
Ok
RUN
Enter N>Ø? 5
Sum of Integers from 1 to  5 = 15
Sum of Squares(Y/N)? Y
Sum of Squares from 1 to  5 = 55
Ok
```

this program calculates the sum of integer numbers from 1 to N (where N is entered from keyboard) and optionally the sum of the square of these numbers. The program has two subroutines; SUB1 and SUB2 written at the end of the program (statements from 5Ø to 1ØØ and from 11Ø to 14Ø).

First of all statements 1Ø, 2Ø and 3Ø are executed. Statement 3Ø (GOSUB) calls the subroutine SUB1 and its statements are executed in sequence up to statement 9Ø. This statement executes a test:

- if X$ (entered from keyboard) is different from "Y", control passes to the statement 1ØØ (RETURN) and then to statement 4Ø (END)

- if X$ equals "Y", control passes to SUB2 ("nested subroutine"). When statement 14Ø (RETURN of SUB2) is reached, control passes to statement 1ØØ (RETURN of SUB1), then to statement 4Ø (END)

ON...GOSUB/RETURN (PROGRAM)

ON...GOSUB calls one of several specified subroutines, depending on the value of a given expression.

RETURN transfers control to the statement following the most recent ON...GOSUB (or GOSUB) that has been executed.

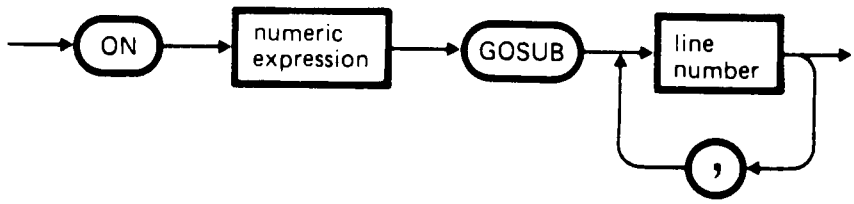


Figure 10-3 ON...GOSUB Statement

Figure 10-4 RETURN Statement

Where

SYNTAX ELEMENT	MEANING
numeric expression	its value determines which line number in the list will be used for branching. A value of 1 causes the subroutine at the first line number in the list to be called; a value of 2 causes the subroutine at the second line number in the list to be called and so on. If the value is a non integer, it is rounded to the nearest integer. If the value is zero or greater than the number of items in the list (but less than or equal to

255), BASIC continues with the next executable statement. If the value is negative or greater than 255, an "Illegal function call" error occurs

line number — each line number in the list must be the first line number of a subroutine

Example

DISPLAY

```
LIST
1Ø INPUT "Enter 1,2,or3";K%
2Ø ON K% GOSUB 4Ø,5Ø,6Ø
3Ø END
4Ø PRINT "SUB1":RETURN
5Ø PRINT "SUB2":RETURN
6Ø PRINT "SUB3":RETURN
Ok
RUN
Enter 1,2,or3? 2
SUB2
Ok
```

COMMENTS

if you enter 1, 2, or 3 the program will display SUB1, SUB2 or SUB3 respectively. In every case a RETURN statement transfers control to the END.

If you enter an integer between Ø and 255, other than 1, 2, or 3, the program will display nothing

PCOS COMMANDS CALLED FROM BASIC AND ASSEMBLY LANGUAGE SUBPROGRAMS

CALL and EXEC allow you to call PCOS commands or Assembly language subprograms, when you are in BASIC.

Both CALL and EXEC statements perform the same function but:

- EXEC is used when the arguments to be passed to the corresponding parameters are constants

- CALL is used when the arguments to be passed to the corresponding parameters are either constants or program variables or both.

CALL and EXEC may be used either in a BASIC program or in immediate mode but they are more often used in a program. At the end of the execution of an Assembly language subprogram or a PCOS command, control returns either to the statement following the call (if CALL or EXEC were used in a program), or to BASIC Command Mode (if CALL or EXEC were used in Immediate Mode).

CALL and EXEC allow a BASIC program to communicate with the PCOS operating system, for example to set system global variables to desired values before other BASIC programs and PCOS commands are executed. At the end of the execution of such a program you may remain in BASIC or go to PCOS (by the SYSTEM Command).

Usually a system initialization program is called INIT.BAS. This is a reserved file name. The M20 system just after loading PCOS and BASIC, searches for that file on both drives. If the file is found, the M20 enters BASIC and INIT.BAS is run.

Remarks

At the end of execution of a CALL or EXEC statement activating a SBASIC PCOS command, the newly set values will not be taken into account in the current program (otherwise the current program could be destroyed).The newly set values will become operative in subsequent programs, thus an EXEC "ba file identifier" often follows an EXEC "sb..." statement.

The EXEC statement only (not the CALL statement) allows you to execute a device re-routing command while in BASIC (for further details see "Professional Computer Operating System (PCOS) - User Guide").

CALL (PROGRAM/IMMEDIATE)

Calls a PCOS command or an Assembly language subprogram, passing either program variables or constant arguments to the subprogram.

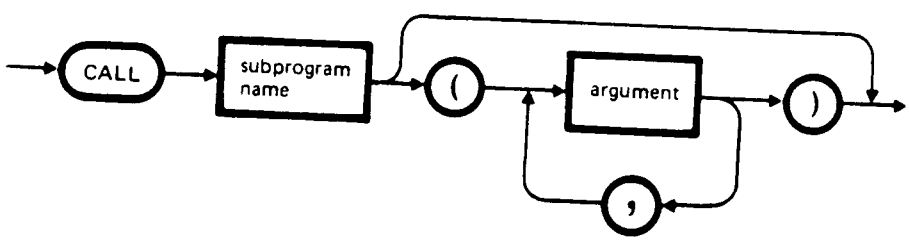


Figure 10-5 CALL Statement

Where

SYNTAX ELEMENT	MEANING
subprogram name	may either be the name of a PCOS command or the name of an Assembly language subprogram. It must be either a string constant or a string variable
argument	may be a constant or a simple variable or an expression whose value is passed to the corresponding parameter (in the same way as an argument is passed to the corresponding parameter in a function call, See Chapter 9). If it is an output argument (i.e. a program variable into which a value is returned), the argument name must be preceded by an "at" sign (@). A variable argument (both an input and an output argument) must be initialized before executing the CALL statement.

Examples

DISPLAY	COMMENTS
1Ø DEFINT A-C . . . 3Ø FILE$="VOL1:FILEØØ1" 4Ø SIZE%=1Ø 5Ø CALL "fn"(FILE$,SIZE%) . . . 9Ø C$="LIST" 1ØØ CALL "pk"(&41,C$) . . . 22Ø A=1Ø 23Ø B=2Ø 24Ø C=2ØØ 25Ø CALL "SUB121"(A,B,@ C) . . .	statement 5Ø calls PCOS command FNEW, passing the file identifier by the string variable FILE$ and the file size by the numeric variable SIZE%. Statement 1ØØ calls the pkey PCOS command, specifying the key by the hexadecimal constant &41 (i.e. A, see Appendix A) and the corresponding string by the variable C$. Statement 25Ø calls the Assembly language subprogram SUB121 specifying two input arguments (A and B) and one output argument (@C). Note that A, B and C have been initialized before.

Remarks

The PCOS command LTERM (Line Terminator) is normally called from BASIC by the CALL statement. It returns an integer (Ø, 1, 2) corresponding to the respective carriage return (↵ , S1 , S2) last entered.

The PCOS command CI (Communication Interface) is normally called from BASIC to send and receive characters to and from a communication RS-232-C port. Other PCOS commands (LABEL, SPRINT, BVOLUME, etc.) are normally called from BASIC.

For more information see "Professional Computer Operating System (PCOS) User Guide" and for CI command see "I/O with External Peripherals User Guide".

EXEC (PROGRAM/IMMEDIATE)

Calls a PCOS command or an Assembly language subprogram passing constant values to the subprogram.

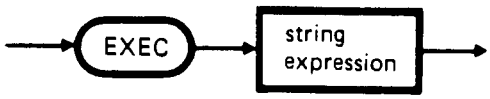


Figure 10-6 EXEC Statement

Where

SYNTAX ELEMENT	MEANING
string expression	its value is interpreted as a subprogram name followed by a list of constant arguments

Remarks

If EXEC calls a PCOS command, the contents of the string expression following EXEC must agree with the command as it would be entered if you were in PCOS.

If EXEC calls an Assembly language subprogram, the contents of the string expression following EXEC is a list of parameters separated by commas. The first of them specifies the subprogram name and the following parameters specify the arguments to be passed to the subprogram.

Note: The arguments are not enclosed in parentheses and may only be constant arguments.

Examples

DISPLAY	COMMENTS
: : 1ØØ EXEC "pk '#', 'RUN V1:CASHFLOW'" : : 15Ø EXEC "fp 1:MY.FILE/SECRET" : : 18Ø A$="fn 1:FILEA,15" : : 23Ø EXEC A$	statement 1ØØ allows you to call the PKEY PCOS command. Note that the strings: - # - RUN V1:CASHFLOW must be surrounded by a pair of single quotes (') as if you were in PCOS. Statement 15Ø allows you to call the FPASS PCOS command. Statement 23Ø allows you to call the FNEW PCOS command, specifying the command as the contents of the string variable initialized in statement 18Ø.

SYSTEM (PROGRAM/IMMEDIATE)

Returns to PCOS and closes all data files.

Figure 10-7 SYSTEM Command

Remarks

SYSTEM allows you to exit BASIC and return to PCOS. It may be used both in a program and an immediate line. SYSTEM is often used at the end of an initialization program which executes a series of PCOS commands and/or Assembly language subprograms using CALL and EXEC statements.

PROGRAMMABLE KEYS

By using the CTRL and COMMAND keys, in conjunction with other non-shift keys, you may assign a special meaning to each key.

This may be a BASIC or PCOS command, an expression, a constant, or any group of characters that you may find useful to have on the keyboard. Assignment can be made either in a BASIC program via the CALL "pk" (or EXEC "pk...") statement, or in PCOS via the PKEY command.

Depending on your needs, assignment of a specific function to a key can be "permanent", automatically made every time you switch on the system (and the system disk is mounted in a drive), or "temporary", to last until the machine is on. For more details see "Professional Computer Operating System (PCOS) User Guide".

11. PROGRAM SEGMENTATION

ABOUT THIS CHAPTER

In this chapter we shall look at the techinque of Program Segmentation and how to pass data from one program to another. We shall illustrate CHAIN (and its several options) and common statements. Moreover, we shall look again at the use of RUN and LOAD with the R option.

CONTENTS

PROGRAM SEGMENTATION

WHEN USING PROGRAM SEGMENTATION

Program segmentation means splitting a large program into two or more smaller programs ("segments") which may be executed in sequence to solve the same problem. Using this technique you may execute programs which could be larger than the available memory, but Program Segmentation is useful in many other situations too (some of these situations are illustrated in the following table).

IF...	THEN...
a program is larger than the available memory	you need to split it into several small programs to be executed one after the other
a program has sections which are rarely executed	you could code these sections as separate programs and load them into memory when necessary
a program has a section which must always be resident, whereas other sections may be transient (and/or used by other programs)	you could code these sections as separate programs. The resident segment (root) will load the first transient segment (overlay) this (or the root) will load the second, and so on Each overlay (or a part of it) may be deleted before a new overlay is loaded
a program may be divided into different sections, each performing a specific function	you could code these sections as separate programs to reduce the cost of programming

PASSING DATA

Program segmentation can imply the need to pass data from one segment to another.

This may be done in several ways as shown in the following table.

IF you use...	THEN...
CHAIN in conjunction with one or more COMMON statements	BASIC creates a "common area" which is not deleted when the CHAINed program is loaded (whereas the current program is deleted). The common area is passed to the CHAINed program and contains all the variables specified in the COMMON statement(s)
CHAIN with ALL	all the variables defined in the current program are passed to the CHAINed program
CHAIN and the current program accesses one or more data files	you may pass data to the CHAINed program via data files. The CHAIN statement does not close data files. Passing data via data files is compatible with: - CHAIN and COMMON statement - CHAIN with ALL - CHAIN with MERGE (and possibly DELETE - see the DELETE option explanation later in this Chapter)
RUN or LOAD with the option R, and the current program accesses one or more data files	you pass data to the specified program via data files. RUN and LOAD with R do not close data files

PROGRAM CHAINING

As we have already seen program segmentation may be performed by the use of:

- CHAIN and COMMON statements

- RUN and LOAD commands.

PROGRAM SEGMENTATION

The CHAIN statement, with its several options, gives you a powerful tool for segmenting a program.

CHAIN may be used either:

- in conjunction with COMMON statements to pass common variables to the CHAINed program, or

- with the MERGE option to merge the CHAINed program with the current one (DELETE is often used in conjunction with MERGE to delete a section of the program, allowing overlays to be loaded in sequence) or

- with the option ALL to pass all the variables to the CHAINed program.

CHAIN (PROGRAM)

Chains a specified program to the program in memory and allows you to pass variables.

CHAIN leaves the files open and preserves the current OPTION BASE setting.

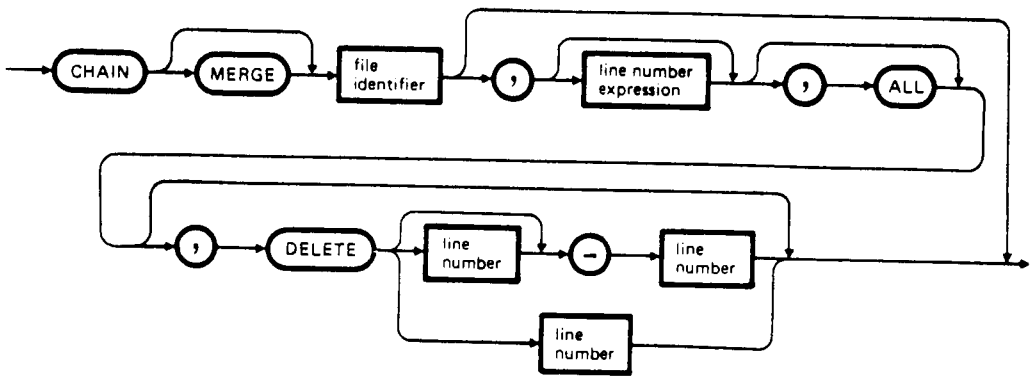


Figure 11-1 CHAIN Statement

Where

SYNTAX ELEMENT	MEANING
MERGE	specifies that the CHAINed program is MERGed with the program in memory. The CHAINed program must be an ASCII file.

IF MERGE is omitted, the program in memory is deleted (except the common area) after the CHAINed program has been loaded.

It is often used with line number expression and DELETE to load overlays (see Examples).

Note: MERGE option preserves variable types for use by the CHAINed program. When using the MERGE option, user-defined functions will be undefined after the merge is complete

file identifier — is a string expression which specifies the program file to be CHAINed

line number expression — is either a line number or an expression that evaluates to a line number in the CHAINed program.

It is the starting point for execution of the CHAINed program.

It is often used with MERGE and DELETE to load overlays.

If it is omitted, execution begins at the first line.

Note: Line number expression is not affected by a RENUM command

ALL — specifies that all the variables of the program in memory are to be passed to the CHAINed program. (It preserves variable types).

If it is omitted, information is passed either by the use of a common area or by the use of data files

DELETE — specifies (by a range of line numbers) that a section of the current program has to be deleted.

The DELETE operation comes before the CHAINed program has been loaded.

DELETE is often used with MERGE and line number expression, to load overlays.

Note: The line numbers used after DELETE are affected by a RENUM command

Examples

DISPLAY	COMMENTS
1Ø REM PROG1 2Ø COMMON A1,B1,C1$ . . . 1ØØ CHAIN "PROG2" 11Ø END	program PROG1 chains PROG2 and passes the values of A1,B1, and C1$ to it (by use of a common area). PROG2 resides on the last selected drive.
1Ø REM PROG2 2Ø COMMON A2$,B2$ . . . 8Ø CHAIN "PROG3",2ØØ 9Ø END	program PROG2 chains PROG3 and passes the values of A2$ and B2$ to it (by use of a common area). The starting point for execution of PROG3 is line 2ØØ. PROG3 resides on the last selected drive
1Ø REM PROG1Ø . . . 5Ø CHAIN "1:PROG11", 1ØØ, ALL 6Ø END	program PROG1Ø chains PROG11 and passes all the program variables to it. The starting point for execution of PROG11 is line 1ØØ.

PROG11 resides on the diskette inserted in drive 1

```
1Ø REM ROOT
 :
 :
1ØØ CHAIN MERGE "V1:OVERLAY1", 1ØØØ
11Ø END
```

ROOT chains OVERLAY1 with the option MERGE. OVERLAY1 must be an ASCII format file residing on the disk named V1. It will be executed starting from line 1ØØØ

```
1ØØØ REM OVERLAY1
 :
 :
15ØØ CHAIN MERGE "V1:OVERLAY2",
     1ØØØ, DELETE 1ØØØ-15ØØ
151Ø END
```

OVERLAY1 chains OVERLAY2 with the option MERGE. OVERLAY2 must be an ASCII format file residing on the disk named V1. Before it is loaded, lines 1ØØØ to 15ØØ will be deleted in memory.

OVERLAY2 will be executed starting from line 1ØØØ

COMMON (PROGRAM)

Defines a common area which is not erased by the CHAINed program and allows you to pass variables from one program to another.

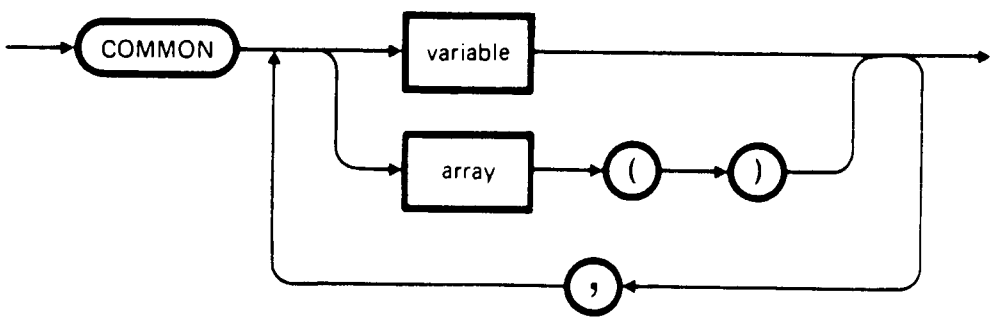


Figure 11-2 COMMON Statement

Examples

DISPLAY

```
1Ø REM PG1
2Ø COMMON A1,B1,C1,D1$
 .
 .
 .
8Ø CHAIN "VOL2:PG2"
9Ø END
```

```
1Ø REM PG2
2Ø PRINT A1,B1,C1,D1$
 .
 .
 .
12Ø END
```

COMMENTS

COMMON statements are used in conjunction with a CHAIN statement.

A program may have one or more COMMON statements.

Variables specified in these statements are allocated in the common area starting from the beginning and in the order in which they appear in the program.

The CHAINed program need not specify, through the use of COMMON statements, the common variables specified by the CHAINing program.

The CHAINed program will use these variables with the same names specified in the CHAINing program.

In our example the values of the variables A1, B1, C1 and D1$ in the program PG1 are passed to the CHAINed program PG2, which may display them (see statement 2Ø).

```
1Ø REM PG1
2Ø DEFDBL C
3Ø COMMON A1,B1,C1,D1$
 .
 .
 .
9Ø CHAIN "VOL2;PG2"
1ØØ END
```

```
1Ø REM PG2
2Ø DEFDBL C
 .
 .
 .
13Ø END
```

Each type definition statement (DEFINT, DEFSNG, DEFDBL, DEFSTR) referring to common variables, must precede the associated COMMON statement and must be repeated in the CHAINed program. (Note the statements DEFDBL, both within PG1 and PG2).

```
10 REM PROGRAM1
20 COMMON A$,B$,C$
30 COMMON A$,A1
 .
 .
100 END
```

it is not good programming practice to repeat a same variable name (in this case A$) either in different COMMON statements of the same program, or in the same COMMON statement. In any case multiple definitions are equivalent to a single definition.

```
10 REM PG1
20 DIM A1(15,20)
30 COMMON A1(),B1,C1
 .
 .
100 CHAIN "VOL2:PG2"
110 END

10 REM PG2
 .
 .
50 PRINT A1(1,1)
 .
 .
90 END
```

a COMMON statement can also specify array names. Such specifications are followed by a pair of parentheses.

Each use of common array must be explicitly described by a DIM statement in the CHAINing program (but not in the CHAINed one, otherwise a "Duplicate Definition" error occurs).

The DIM statement must be written before the associated COMMON statement.

```
10 REM mod1
20 A=1:B=2
30 COMMON A,B
40 GOTO 60
50 COMMON C
60 CHAIN "mod3"

10 REM mod2
20 A=1:B=2
30 COMMON A
40 GOTO 60
50 COMMON B
60 CHAIN "mod3"

10 REM mod3
20 PRINT A;B
```

The COMMON statement is a declarative statement, thus it allocates a common area even if control of execution does not pass through it.

For example, when executing program "mod1" an "Illegal function call in 50" is issued, as variable C has not been initialized. When executing program "mod2" instead, program "mod3" is CHAINed: it displays both A and B variables, even if statement 50 of "mod2" is jumped over.

Remark

Common variables must always be initialized within the CHAINing program.

12. DISK FILE HANDLING

ABOUT THIS CHAPTER

This chapter describes the two types of external data files available; sequential and random files. We shall see how each is created, opened and closed and how to get data in and out of them.

CONTENTS

DISK FILE HANDLING

SEQUENTIAL AND RANDOM FILES

A data file is created (i.e. made known to the system) either by:

- the PCOS command FNEW which gives a name to a new file and specifies its initial size

- the OPEN statement which allows a BASIC program to access the file.

OPEN gives a name to a file (which has not yet been created by FNEW or another OPEN). Moreover it associates a data buffer with the file (to be used for any Input/Output operation) and specifies an access mode.

If you must create a very large data file and you know the final file size fairly accurately, then create the file by FNEW instead of by an OPEN statement. FNEW will allocate a sequence of contiguous disk sectors to the file thus making Input/Output operations more efficient. Moreover FNEW will assure you that there is enough room for the file on the disk.

All files are "byte stream" only, and thus have no intrinsic data format or data interpretation upon I/O. There are four possible modes to open a file in, however. These modes control only the type of access that will be allowed, and do not add any interpretation of the data flow.

The access mode may be changed for a file each time it is re-OPENed.

The table below summarizes the main features of a data file and classifies files into two categories (sequential and random) depending on the access mode used.

FILE TYPE	CHARACTERISTICS	ACCESS MODE
Sequential (or Stream-oriented)	a sequential file is considered as a sequence of ASCII characters without any grouping criterion. The number of data items read or written by each Input/Output statement can vary and is usually determined by the list of variables specified in the statement	Input: sequential input (one item after another) from the beginning of the file Output: sequential output from the beginning of the file. Data on the file (if any) is lost

		Append: sequential output from the end of the file. Data on the file is not lost
Random (or Record-oriented)	a random file is considered as a sequence of data grouped in records. Each Input/Output statement may read or write one record at a time. The records of a random file all have the same length and structure	Random: direct access Input/Output to the specified record

SEQUENTIAL FILES

Sequential files are the simplest way to store data. They are ideal for storing free-form data (which may not be grouped in records). The data that is written to a sequential file is stored, one item after another (sequentially), in the order it is sent and is read back in the same order.

There are several points to bear in mind:

- if you open a sequential file in Output, you start writing at the beginning of the file and the file's previous contents are lost

- if you open a sequential file in Append, you start writing after the last data item on the file

- to update a sequential file, open the file in Input, read the file and write the updated data to a new file which must have been opened in Output

- data written on a sequential file usually includes delimiters to signify where each data item begins and ends

- to read a sequential file, you must open it in Input and you must know the format of the data; whether for example, the file consists of numbers separated by blanks, or of numbers and strings separated by commas

- a data item on the file is always written as a string of characters (one byte for each character of data). For example, the number:

  351.27

  requires 6 bytes of disk storage, excluding the delimiters (which may be blanks or commas).

RANDOM FILES

These are ideal for storing data which may be grouped in records. The records of a random file must all be of the same length.

Accessing random files requires more program steps than sequential files but there are advantages when using random files:

- instead of having to start reading or writing at the beginning of a file, you can read or write any record you specify

- to update a file, you do not have to read the entire file, update the data and write it again. You can rewrite or add to any record you choose, without accessing all the preceeding records

- opening a random file allows you to read and write from the file via the same buffer.

OPENING AND CLOSING FILES

To access a file with a BASIC program, you must open it with an OPEN statement. This specifies the file identifier, the access mode, the file number and if the file is a random file, the record length.

The maximum number of concurrent files (i.e., OPENed at the same time may be set by the PCOS command SBASIC or assumed by default (the default value is 3). The maximum number cannot exceed 15.

Whenever you open a file, a file (or buffer) number is associated with the file. Each buffer is given a number from 1 to 15. You will use this number to specify the file in any I/O statement of your program. You can think of a buffer as a waiting area that data must pass through on the way to and from the disk file.

For random files, the user must define the structure of the buffer (i.e., of the records in the file) by fixing the length (in characters) of each data item within the buffer by a FIELD statement.

When you access a file by an Input/Output statement, you must specify the file by its file number instead of its identifier.

When you CLOSE a file you delete the connection between the file and its buffer and that file may no longer be accessed, until you re-OPEN it. If you re-OPEN it, you may associate either the same or another buffer with the file.

OPEN (PROGRAM/IMMEDIATE)

Opens a disk file allowing Input/Output operations on the file.

If the specified file is not found it will be created (unless access mode is "I" - See Remark below).

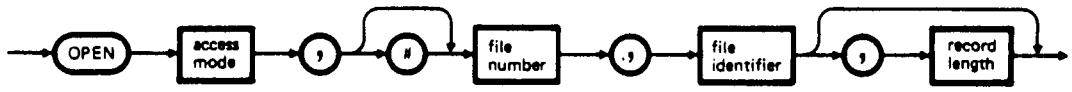


Figure 12-1 OPEN Statement

Where

SYNTAX ELEMENT	MEANING
access mode	is either a string constant or a string variable containing one of the following characters:

- "A" (Append): sequential output after the last data item on a sequential file. Data in the file (if any) is not lost, new data will be added at the end

- "I" (Input): sequential input starting from the beginning of a sequential file

- "O" (Output): sequential output starting from the beginning of a sequential file. Data in the file (if any) is lost

- "R" (Random): Input/Output access to the records of a random file

Note: If a sequential file is empty (i.e. does not contain data), "O" and "A" are equivalent

file number

is a numeric expression whose value, rounded to the nearest integer, must be in the range 1 to 15. The specified file number remains associated with the file as long as it is open and will be used to specify the file in any I/O statement in the program

file identifier

is either a string constant or a string variable and may specify:

- a new file (i.e. unknown to the system); in this case the file is created (except for access mode "I")

- an existing file; in this case the file is only OPENed

record length

is a numeric expression (rounded to the nearest integer) which, if included, sets the record length of a random file.

This parameter may only be set for random files. Its default value is 256 bytes.

Its maximum value is that of the record size parameter set by the PCOS command SBASIC. SBASIC can set the record size parameter from 1 to 1Ø96 (with a default value of 256)

Examples

DISPLAY

```
    .
    :
5Ø OPEN "A",1,"V1:EXAMPLE"
    .
    :
16Ø OPEN "O",2,"V1:TEST"
    .
    :
27Ø OPEN "R",3,"V2:F1",8Ø
28Ø OPEN "R",4,"V2:F2",2Ø
    .
    :
49Ø CLOSE 2
5ØØ OPEN "I",5,"V1:TEST"
    .
    :
6ØØ OPEN "R",2,FILE$,RN
```

COMMENTS

Statement 5Ø opens the sequential file EXAMPLE, which is resident on the disk named V1. The access mode is Append and file number 1 is associated with the file.

Statement 16Ø opens the sequential file TEST, which is resident on the disk named V1. The access mode is Output and file number 2 is associated with the file.

Statement 27Ø opens the random file F1, which is resident on the disk named V2. The file number 3 is associated with the file and a record length of 8Ø bytes is set.

Statement 28Ø opens the random file F2, which is resident on the disk named V2. The file number 4 is associated with the file and a record length of 2Ø bytes is set.

Statement 49Ø closes the file TEST.

Statement 5ØØ re-opens the file TEST in Input mode and associates the file number 5 with it.

Statement 6ØØ opens a random file, whose identifier is the contents of the string variable FILE$. The record length is the contents of the numeric variable RN. The associated file number is 2. It has been made available by statement 49Ø

Remark

You cannot create a file by an OPEN statement if you specify "I" as access mode. If you try to, a "File not found" error occurs.

CLOSE (PROGRAM/IMMEDIATE)

Closes disk files.

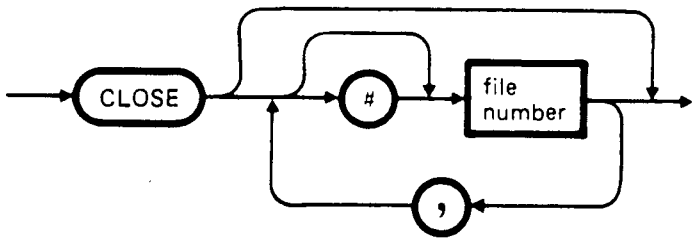


Figure 12-2 CLOSE Statement

Where

SYNTAX ELEMENT	MEANING
file number	is a numeric expression whose rounded value specifies the number of the buffer associated with the file. This number must be from 1 to 15. A CLOSE with no parameters closes all open data files

Examples

DISPLAY	COMMENTS
: : 17Ø CLOSE #2	Statement 17Ø closes the file whose file number is 2.
. 25Ø A=6	Statement 29Ø closes the files whose file numbers are 3,5 and 6 (if A equals 6).
. 29Ø CLOSE 3,5,A	
: : 12ØØ CLOSE	Statement 12ØØ closes all the files

Characteristics

IF...	THEN...
a CLOSE is executed	the association between a file and its buffer is deleted; that buffer may now be reused to OPEN any file. A CLOSEd file may be re-OPENed by another OPEN statement (within the same or another program) and any free buffer may be associated with the file
an END statement or a SYSTEM command is executed	all OPENed data files are CLOSEd
a CTRL RESET is issued	all OPENed data files are CLOSEd, and any data still in buffers, and not yet written to disk will be lost
any modification is made to the current program (line insertion, line editing and so on...)	all OPENed data files are CLOSEd
either a CHAIN statement or a LOAD (RUN) command with the option R is executed	no OPENed data files are CLOSEd

a program interruption occurs (upon execution of a STOP statement, or when an error message is issued, or when the user presses CTRL C)	no OPENed data files are CLOSEd
an attempt is made to CLOSE an already CLOSEd or not yet OPENed file	the CLOSE statement has no effect

Remark

It is good programming practice to always CLOSE a file when you have finished with it, unless you want to chain another program (by CHAIN or RUN with the R option or LOAD with the R option) working on the same files and with the same acces mode. A LOAD or RUN without the R option, or a SAVE command close all open files.

WRITING A SEQUENTIAL FILE

To write a sequential file you must OPEN it in Output ("O") or Append ("A").

Output statements are PRINT#, PRINT# USING and WRITE#.

PRINT# and WRITE# output standard format data, whereas PRINT # USING outputs data in a user defined format.

The difference between PRINT# and WRITE# is that:

- PRINT# writes data to a disk in the same format used by the PRINT statement

- WRITE# writes data to a disk in the same format used by the WRITE statement, i.e. inserting commas between data and quoting string values.

Note: LOC function may be used to know the number of sectors (256 byte blocks) written to or read from the a file since it was OPENed, to avoid a "Disk full" error message.

The following steps are required to write data to a sequential file.

STEP	OPERATION	EXAMPLES
1	Open the file, specifying either "A", or "0" as access mode	1Ø OPEN "0",1,"1:F1"
2	Write a series of numeric and/or string values to the file, using an output statement	5Ø WRITE#1,A$,B,C$
3	Repeat step 2 for each output operation	15Ø WRITE#1,A1,B1,C1$ 18Ø WRITE#1,A2,B2$,C2,D2
4	When you have finished with the file close it (unless another CHAINed program uses the file with the same access mode)	3ØØ CLOSE#1

PRINT # (PROGRAM/IMMEDIATE)

Writes data to a sequential file, in the same way as the PRINT statement.

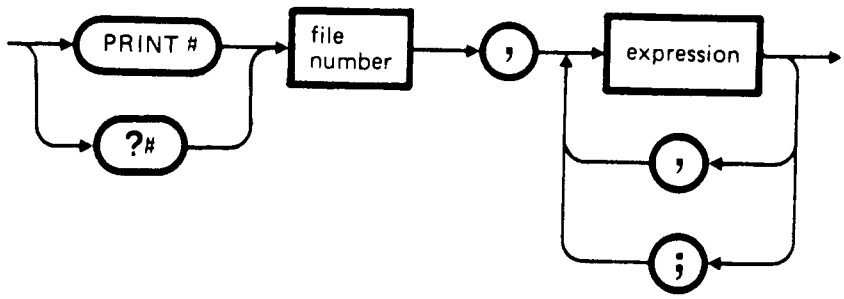


Figure 12-3 PRINT# Statement

Where

SYNTAX ELEMENT	MEANING
file number	is a numeric expression whose rounded value specifies the number of the buffer associated with the file
expression	is a numeric, relational, logical or string expression whose value is written to the file

Remark

An image of the data is written to the disk, just as it would be displayed on the screen with a PRINT statement. For this reason, care should be taken to delimit the data on the disk, so that it will be input correctly from the disk.

Characteristics

IF...	THEN...
a PRINT# statement is executed	data is output sequentially to the specified file

the file is OPENed for Output ("O")	the file pointer is set to the beginning of the file, therefore your first PRINT# places data at the beginning of the file. For each PRINT# operation, the pointer advances, so the values are written in sequence
the file is OPENed for Append ("A")	the file pointer is set to the end of the file, therefore, your first PRINT# places data after the last data item on the file. For each PRINT# operation the pointer advances, so the values are written in sequence
you want to set up your PRINT# list correctly for access by one or more INPUT# statements	remember that a PRINT# statement creates a disk image similar to that which a PRINT creates on the screen. PRINT# writes an ASCII coded image of the data. The punctuation in the PRINT# list is very important. Unquoted commas and semicolons have the same effect as they do in PRINT statements
you have to output numeric values (resulting from the evaluation of a numeric, relational or logical expression)	you may use both commas or semicolons to separate the expressions. Generally you would not want to waste disk space, so you should use semicolons instead of commas. For example:

```
LIST
1Ø OPEN "O",#1,"DATA1"
2Ø A=1:B=2:C=3
3Ø PRINT#1,A;B;C
4Ø CLOSE#1
5Ø OPEN "I",#1,"DATA1"
6Ø INPUT#1,A1,B1,C1
7Ø PRINT A1;B1;C1
8Ø CLOSE#1
9Ø END
Ok
RUN
 1  2  3
Ok
3Ø PRINT#1,A,B,C
RUN
 1  2  3
Ok
```

If you separate the variables A,B and C in statement 3Ø with commas instead of semicolons the program displays the same results but you waste disk space.

With semicolons the disk image will be:

```
 1  2  3
```

With commas it will be:

```
 1             2             3
```

you have to output string values

you have to insert explicit delimiters, if you want to INPUT# them as distinct strings

you have to output string values which do not contain commas, semicolons, significant leading or trailing blanks, carriage returns or line feeds

use a comma as a string constant (",") to separate string expressions in the PRINT# statement. Thus data items will be separated on the disk by a comma and will be read back as different strings by an INPUT#statement.

For example:

```
LIST
1Ø OPEN "O",#1,"DATA1"
2Ø A$="CAMERA"
3Ø B$="936Ø5-2"
4Ø PRINT#1,A$;B$
5Ø CLOSE#1
6Ø OPEN "I",#1,"DATA1"
7Ø INPUT#1,A1$
8Ø PRINT A1$
9Ø CLOSE#1
1ØØ END
Ok
RUN
CAMERA936Ø5-2
Ok
4Ø PRINT#1,A$;",";B$
7Ø INPUT#1,A1$,B1$
8Ø PRINT A1$,B1$
RUN
CAMERA        936Ø5-2
Ok
```

If you separate A$ and B$ by a semicolon in statement 4Ø, the disk image will be:

```
CAMERA936Ø5-2
```

Because there are no delimiters this cannot be input as two separate strings. To correct the problem, insert an explicit delimiter (",") into statement 4Ø and modify statements 7Ø and 8Ø too. The disk image will be:

```
CAMERA,936Ø5-2
```

This can be read back into two string variables (see the new run)

you have to output string values containing commas, semicolons, significant leading or trailing blanks, carriage returns or line feeds

write them to disk and surround them by explicit quotation marks, CHR$(34).

For example:

```
LIST
1Ø OPEN "O",#1,"DATA1"
2Ø A$="CAMERA, AUTOMATIC"
3Ø B$="   936Ø5-2"
4Ø PRINT#1,A$;B$
5Ø CLOSE#1
6Ø OPEN "I",#1,"DATA1"
7Ø INPUT#1,A$,B$
8Ø PRINT A$;B$
9Ø CLOSE#1
1ØØ END
Ok
RUN
CAMERAAUTOMATIC   936Ø5-2
Ok
4Ø PRINT#1,CHR$(34);A$;CHR$(34);
   CHR$(34); B$;CHR$(34)
RUN
CAMERA, AUTOMATIC   936Ø5-2
Ok
```

Statement 4Ø writes the following image to disk:

```
 CAMERA, AUTOMATIC   936Ø5-2
```

and statement 7Ø inputs

```
CAMERA
```

to A$ and

```
AUTOMATIC   936Ø5-2
```

to B$, as you can check by statement 8Ø, when you run the program for the first time. If you change statement 4Ø as indicated, you write the following image to disk:

"CAMERA, AUTOMATIC"" 9360̸5-2"

and statement 7̸0̸ inputs

"CAMERA, AUTOMATIC" to A$ and
" 9360̸5-2" to B$, as you can check by statement 8̸0̸, when you run the program for the second time

PRINT# USING (PROGRAM/IMMEDIATE)

Writes data to a sequential file in a user defined format in the same way as PRINT USING statement displays data on the screen.

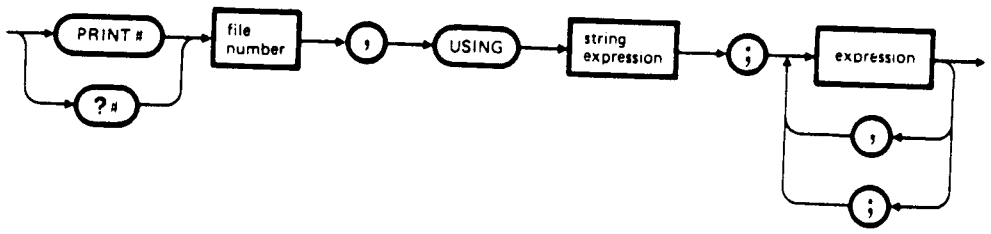


Figure 12-4 PRINT# USING Statement

Where

SYNTAX ELEMENT	MEANING
file number	is a numeric expression whose rounded value specifies the number of the buffer associated with the file
string expression	is the formatting characters fully described in Chapter 7
expression	is a numeric, relational, logical, or string expression to be written to the file

Remarks

Care should be taken to delimit data items on the disk, so that they will be input correctly by an INPUT# statement.

For example, the statement:

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

See Chapter 7 for full details of the facilities offered by the PRINT USING statement.

WRITE# (PROGRAM/IMMEDIATE)

Writes data to a sequential file, in the same way as the WRITE statement displays data on the screen. Each data item will be separated from the preceding one by a comma. Strings will be delimited by quotation marks ("). After the last item in the list is written to disk, BASIC inserts a carriage return/line feed.

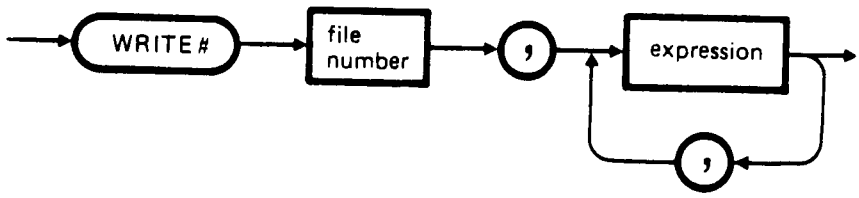


Figure 12-5 WRITE# Statement

Where

SYNTAX ELEMENT	MEANING
file number	is a numeric expression whose rounded value specifies the number of the buffer associated with the file

expression	is a numeric, relational, logical or string expression to be written to the file

Remarks

It is not necessary to put explicit delimiters in the list of a WRITE statement

If you want to write a string to a disk file that contains a quotation mark ("), you must use a PRINT# instead of a WRITE# statement. A quotation mark may be inserted by the CHR$(34) in a string value which does not contain commas, semicolons, significant leading or trailing blanks, carriage returns or line feeds. A quotation mark may also belong to a string variable whose value is assigned by use of the READ and DATA statements, or by an INPUT (LINE INPUT, INPUT#, LINE INPUT#) statement.

Example

DISPLAY

```
LIST
1Ø OPEN "O",1,"DATA2"
2Ø A$="CAMERA"
3Ø B$="936Ø5-2"
4Ø WRITE 1,A$,B$
5Ø CLOSE 1
6Ø OPEN "I",1,"DATA2"
7Ø INPUT 1,A$,B$
8Ø WRITE A$,B$
9Ø CLOSE 1
1ØØ END
Ok
RUN
"CAMERA","936Ø5-2"
Ok
```

COMMENTS

Statement 4Ø writes the following image to disk:

"CAMERA","936Ø5-2"

Statement 7Ø inputs "CAMERA" to A$ and "936Ø5-2" to B$, as you can check by statement 8Ø

LOC

With sequential files, LOC returns the number of sectors (256 byte blocks) read from, or written to the file, since it was OPENed.

LOC function may also be used with random files (see below).

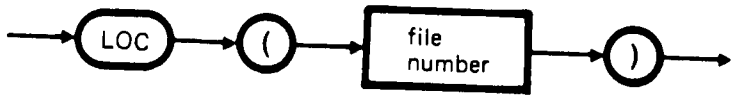


Figure 12-6 LOC Function

Where

SYNTAX ELEMENT	MEANING
file number	is a numeric expression rounded to the nearest integer. It is the number of the buffer associated with the file

Example

```
200 IF LOC(2)>30 THEN STOP
```

READING A SEQUENTIAL FILE

To read a sequential file, you must open it in Input mode ("I").

INPUT# and LINE INPUT# statements allow you to read data from a sequential file. INPUT# reads one or more data items separated by delimiters and assigns them to numeric and or string variables. LINE INPUT# reads an entire line and assigns it to a string variable.

Besides these two statements, BASIC allows you to use the following two functions, which are very useful in handling sequential files:

- the EOF function which allows you to test whether an end of file condition exists to avoid further read operations which would cause the following message to appear:

Input past end

- the LOC function which tells you the number of sectors (256 byte blocks) read from or written to the file, since it was OPENed.

The following program steps are required to read data from a sequential file.

STEP	OPERATION	EXAMPLES
1	open the file, specifying "I" as access mode	1Ø OPEN "I",#2,"DATA"
2	input a series of numeric and/or string values from the file, using an INPUT# and/or a LINE INPUT# statement	5Ø INPUT#2,X$,Y$,Z
3	repeat step 2 for each input operation (possibly testing for End Of File)	1ØØ INPUT#2,X1,X2,X3,X4 15Ø INPUT#2,U$,W$
4	when you have finished with the file, close it (unless another CHAINed program uses the file with the same access mode)	2ØØ CLOSE#2

INPUT# (PROGRAM/IMMEDIATE)

Reads data items from a sequential file and assigns them to program variables.

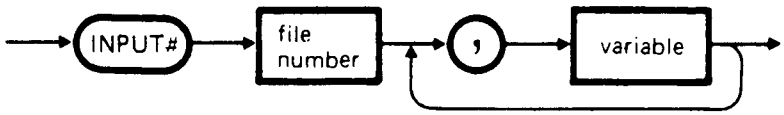


Figure 12-7 INPUT# Statement

Where

SYNTAX ELEMENT	MEANING
file number	is a numeric expression whose rounded value specifies the number of the buffer associated with the file
variable	is the name of a variable which will receive a data item from the file

Remark

Unlike INPUT, the INPUT# statement does not display a prompt (?) when it is executed.

Characteristics

IF...	THEN...
an INPUT# statement is executed	data is input sequentially from the specified file. That is, when the file is first opened, a pointer is set to the beginning of the file. Each time a data item is input, the pointer moves to the next data item. To restart reading from the beginning of the file, close the file and re-open it

you want to input data successfully	you need to know the type (numeric or string) of each successive data item on the file. Data items must be separated by delimiters (see below) Note: Numeric items may be input into string variables. If you input a number into a string, use the VAL function to get the numeric value, to prevent mismatched type errors.
BASIC is inputting to a numeric variable	leading spaces, carriage returns and line feeds are ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed or comma. Note: Numeric conversions are valid. That is a numeric constant may be assigned to a numeric variable of different type, as with a LET, an INPUT or a READ statement (see Chapter 5)
BASIC is inputting to a string variable	leading spaces, carriage returns and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item
the first character is a quotation mark (")	the string item will consist of all characters read between the first quotation mark and the second. The quotation marks themselves do not become a part of the string. (Thus, a quoted string may not contain a quotation mark as a character)

the first character <u>is not</u> a quotation mark	the string is an unquoted string and will terminate with a comma, or carriage return, or line feed (or after 255 characters have been read). For example, if the data on disk is: SUBROUTINES,SUBPROGRAMS "HOW TO CALL THEM?" the statement: INPUT#1,R$,S$,T$ will assign values as follows: R$ = SUBROUTINES S$ = SUBPROGRAM "HOW TO CALL THEM?" T$ = null string If you insert a comma on the disk file before the first quotation mark, i.e. SUBROUTINES,SUBPROGRAMS, "HOW TO CALL THEM?" the same INPUT# statement will assign: R$ = SUBROUTINES S$ = SUBPROGRAM T$ = "HOW TO CALL THEM?"

LINE INPUT# (PROGRAM/IMMEDIATE)

Reads an entire line (up to a carriage return) from a sequential file and assigns it to a string variable.

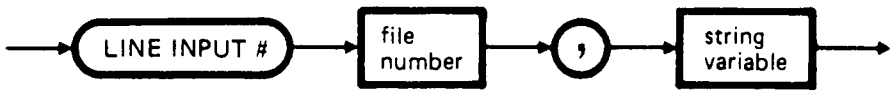


Figure 12-8 LINE INPUT# Statement

Where

SYNTAX ELEMENT	MEANING
file number	is a numeric expression whose rounded value specifies the number of the buffer associated with the file
string variable	is the variable name to which the line will be assigned

Characteristics

IF...	THEN...
a LINE INPUT# statement is executed	a line of string data is read into the specified string variable. LINE INPUT# reads all characters in the file up to: - a carriage return, or - a carriage return/line feed, or - the end of file, or - the 255th data character (this 255 character is included in the string)

leading characters or other delimiters are encountered - quotation marks, commas, blanks, and so on..	they are included in the string
you want to read in data without following the usual restrictions regarding leading characters and terminators	use LINE INPUT# statements
you want to read an ASCII - format BASIC program file as data	use LINE INPUT# statements. (You can write programs that edit other ASCII programs; renumber them, change LPRINTs to PRINTs, etc.)

Remarks

LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence and the next LINE INPUT# reads all characters up to the next carriage return (If a line feed/carriage return sequence is encountered, it is preserved).

Example

DISPLAY

```
LIST
1Ø INPUT "PROGRAM IDENTIFIER";P$
2Ø OPEN "I",1,P$
3Ø K%=Ø
4Ø IF EOF(1) THEN 8Ø
5Ø K%=K%+1
6Ø LINE INPUT# 1,A$
7Ø GOTO 4Ø
8Ø PRINT P$ " IS" K% "LINES LONG"
9Ø CLOSE
1ØØ GOTO 1Ø
11Ø END
Ok
RUN
```

COMMENTS

this program counts the number of lines in an ASCII format program file. Each line ends with a carriage return/line feed, thus the LINE INPUT# in line 6Ø reads one entire line at a time, into the dummy variable A$. Variable K% counts the lines of the program

```
PROGRAM IDENTIFIER? V1:P1
V1:P1 IS 35Ø LINES LONG
PROGRAM IDENTIFIER? V1:P2
V1:P2 IS 152Ø LINES LONG
PROGRAM IDENTIFIER?^C
Break in 1Ø
Ok
```

EOF

Returns -1 (true) if the end of a sequential file has been reached.

Use EOF to test for end of file while INPUTting, to avoid "Input past end" errors.

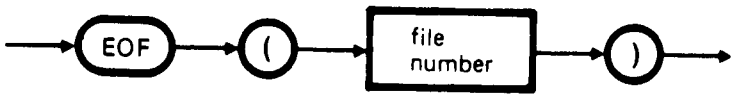


Figure 12-9 EOF Function

Where

SYNTAX ELEMENT	MEANING
file number	is a numeric expression rounded to the nearest integer. It is the number of the buffer associated with the file

Example

```
1Ø DIM A(5Ø)
2Ø OPEN "I",1,"DATA1"
3Ø FOR K%=Ø TO 5Ø
```

```
4Ø IF EOF(1) THEN 1ØØ
5Ø INPUT#1,A(K%)
6Ø NEXT K%
    :
    :
```

UPDATING A SEQUENTIAL FILE

To update a sequential file, read in the file and write out the updated data to a new output file, as indicated by the following table.

STEP	OPERATION
1	Open the sequential file to be updated for Input
2	Open another new sequential file for Output
3	Input a list of data and update them as necessary
4	Output the updated data to the new file
5	Repeat steps 3 and 4 until all data has been read, updated and output to the new file; then go to step 6
6	Close both files (unless you want to chain a program working on the same files with the same access mode)

DEFINING A RECORD LAYOUT

After opening a random file you have to define the record layout by a FIELD statement. FIELD organizes the random file buffer so that you can pass data from the program to disk and vice versa. The record can be divided up into any number of fields by a FIELD statement, but the total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. (The default record length is 256).

The FIELD statement sets up the size of each of these fields and allows string variable names to point to each field. These field names, unlike ordinary strings which point to an area in memory called "string space", point to the buffer area associated with the file.

All data, both strings and numbers, must be placed into the buffer in string form. There are three pairs of functions (MKI$/CVI, MKS$/CVS, MKD$/CVD) for converting numbers to strings and vice versa.

Note: Do not use a field name in an INPUT statement, or on the left side of a LET statement. That name will no longer point to the buffer field (but to the string space); therefore, you will not be able to access that field using the previously assigned field name.

FIELD (PROGRAM/IMMEDIATE)

Defines fields in a random file buffer.

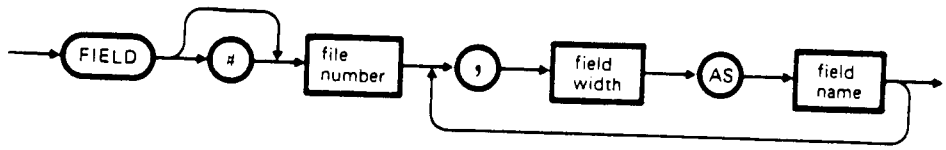


Figure 12-10 FIELD Statement

Where

SYNTAX ELEMENT	MEANING
file number	is a numeric expression whose rounded value specifies the number of the buffer associated with the file
field width	is the number of bytes to be allocated to the field. One byte corresponds to one characters of data
field name	is the string name to be assigned to the field defined by the immediately preceeding field width

Examples

DISPLAY	COMMENTS
2Ø FIELD#1,15 AS NAME$,2Ø AS C$, 1Ø AS P$: : 8Ø NAME$=B$ (Wrong) : : 1ØØ LSET NAME$=B$ (Right)	Statement 2Ø allocates the first 15 positions (bytes) of the random file buffer#1 to the name NAME$, the next 2Ø to C$ and the (last) 1Ø to P$. After executing statement 8Ø NAME$ becomes an ordinary string variable name. You will not be able to access the first field of the buffer any more. Use statement 1ØØ instead (see LSET/RSET statements below)
3Ø FIELD#2,128 AS N1$,128 AS N2$: : 1ØØ FIELD#2,128 AS N3$,1ØØ AS N4$, 28 AS N5$	You may use FIELD any number of times to "re-organize" a file buffer. Re-organizing a buffer by a FIELD statement does not clear the contents of the buffer; only the means of accessing the buffer (the field names) are changed. Thus two or more field names can reference the same area of the buffer
5Ø FIELD#3,16 AS K$(1),112 AS L$(1) : : 9Ø FIELD#3,128 AS DUMMY$, 16 AS K$(2),112 AS L$(2)	You may use a dummy variable in a FIELD statement to "pass over" a portion of the buffer and start fielding it somewhere in the middle. In the second FIELD statement, DUMMY$ serves to move the starting position of K$(2) to position 129

Remarks

It is good programming practice that the sum of all the field widths equals the record length specified by the OPEN statement. In any case

this sum must not be greater than the record lenght, otherwise a "Field overflow" error occurs.

WRITING RECORDS TO A RANDOM FILE

To write records to a random file, you must open it, specifying "R" as access mode.

The PUT-File statement allows you to write a record to a random file. The contents of the record must have been prepared within the random buffer before executing the PUT-File statement by LSET or RSET statements. LSET and RSET move data from memory to the random file buffer by allocating string expressions to the field names previously defined.

If the string expression uses less bytes than you had allocated in the FIELD statement the extra space allocated is padded with blanks. These blanks can be set to be on the left or the right of the string expression value. Left justification (see the LSET statement) starts at the first position of the field. Right justification (see the RSET statement) finishes at the last position of the field. When you have to transfer numeric values into the buffer you must convert them to strings by the MKI$, MKS$ and MKD$ functions.

Note: The LOC function either returns the record number written from a PUT-File statement or gets the record number just read from a GET-File statement.

The following program steps are required to write records to a random file.

STEP	OPERATION	EXAMPLE
1	open the file, specifying "R" as access mode and (optionally) the record length	1Ø OPEN "R",#1,"1:DIR",22
2	field the buffer	2Ø FIELD#1,15 AS A$,5 AS B$, 2 AS C$

3	insert data into the buffer	100 LSET A$="JOHN JONES" 110 LSET B$="U.K." 120 LSET C$=MKI$(I%)
4	write a record to the file	130 PUT 1,5
5	to write another record continue at step 3. Otherwise, go to step 6	
6	close the file (unless you want to chain a program working on the same file with the same access mode)	150 CLOSE#1

LSET/RSET (PROGRAM/IMMEDIATE)

LSET stores a string value in a random buffer field left justified, or left justifies a string value in a string variable

RSET stores a string value in a random buffer field right justified, or right justifies a string value in a string variable.

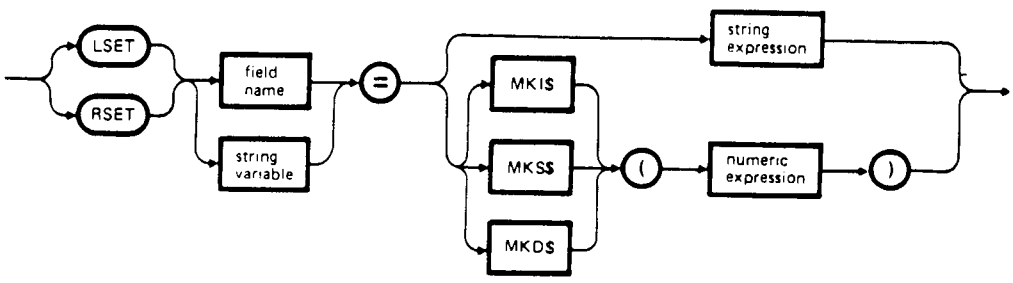


Figure 12-11 LSET/RSET Statements

Where

SYNTAX ELEMENT	MEANING
field name	is a string variable name which specifies the name of a field of a random buffer
string variable	the name of an ordinary string variable
MKI$/MKS$/MKD$	the 'make' function which converts an integer (MKI$), or a single (MKS$), or a double (MKD$) precision value to a string value
string expression	the string to be left or right justified in a given field
numeric expression	the numeric value to be converted to a string and left or right justified in a given field

Examples

```
DISPLAY

1Ø OPEN "R",#1,"1:MYFILE/MYPASS",2Ø
2Ø FIELD#1,1Ø AS N1$,1Ø AS N2$
3Ø LSET N1$="CHARLES"
4Ø LSET N2$="JAMES"
    .
    .
    .
1ØØ RSET N1$="CHARLES"
11Ø RSET N2$="JAMES"
    .
    .
    .
2ØØ LSET N1$="CHARLES THOMSON"
    .
    .
    .
```

COMMENTS

Statements 3Ø and 4Ø put the data in the buffer #1 as follows:

N1$

CHARLES

N2$

JAMES

Statements 1ØØ and 11Ø put the data in the buffer as follows:

N1$

CHARLES

N2$

Statement 2ØØ put the data in the buffer as follows:

N1$

Note: If a string is too long to fit in the specified buffer field, it is truncated on the right, irrespective of whether LSET or RSET was specified.

```
11Ø A$=SPACE$(2Ø)
12Ø RSET A$=N$
```

LSET and RSET can also be used with a non field variable to left justify or right justify a string in a given field. This can be a useful formatting technique when printing output.

In the example on your left RSET right justifies the string N$ in a 2Ø-character field

MKI$/MKS$/MKD$

These functions change a number to a string.

MKI$ converts an integer to a 2-character string

MKS$ converts a single precision value to a 4-character string

MKD$ converts a double precision value to an 8-character string

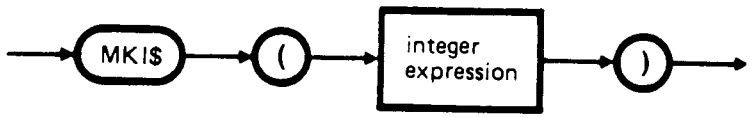


Figure 12-12 MKI$ Function

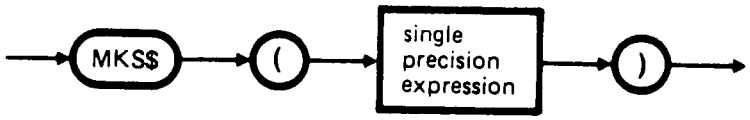


Figure 12-13 MKS$ Function

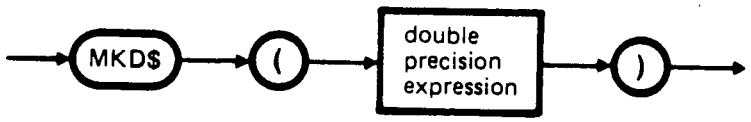


Figure 12-14 MKD$ Function

Examples

DISPLAY	COMMENTS
3Ø LSET D$=MKI$(I%)	Field name D$ would now contain a two byte representation of the integer I%

```
100 STR4C$=MKS$(SPV)
```

A "make" function is not confined to use with the LSET and RSET statements. Here SPV is the name of a single precision variable, which is converted into a 4 character string and assigned to the STR4C$ variable

PUT-File (PROGRAM/IMMEDIATE)

Writes data from a random file buffer to a random file.

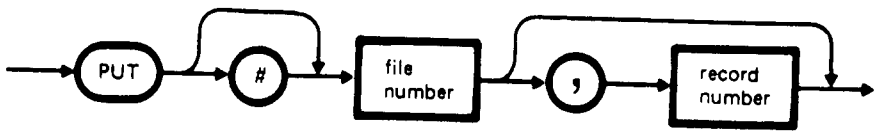


Figure 12-15 PUT-File Statement

Where

SYNTAX ELEMENT	MEANING
file number	is a numeric expression which specifies the number of the buffer associated with the file
record number	is a numeric expression which specifies the record number in the file. The smallest record number is 1, the largest 32767. If this parameter is omitted, the current record number is assumed. Note: The current record is the record whose number is one higher than that of the last record accessed. The first time you access a random file the current record number is set equal to 1

Example

DISPLAY

```
LIST
1Ø OPEN "r",1,"1:RAND",48
2Ø FIELD 1,2Ø AS R1$,2Ø AS R2$,8 AS R3$
3Ø FOR L=1 TO 4
4Ø INPUT "name";N$
5Ø INPUT "address";M$
6Ø INPUT "phone";P#
7Ø LSET R1$=N$
8Ø LSET R2$=M$
9Ø LSET R3$=MKS$(P#)
1ØØ PUT 1,L
11Ø NEXT L
12Ø CLOSE 1
13Ø END
Ok
RUN
name? super man
address? USA
phone? 11234621
name? robin hood
address? England
phone? 23462101
    :
    :
Ok
```

COMMENTS

Statement 1Ø opens the random file RAND, with a record length of 48 on the diskette mounted in drive 1. The file number is 1. Statement 2Ø divides the buffer into fields.

Statement 1ØØ writes a record to file RAND, with the record number being set by the control variable of the FOR/NEXT loop

LOC

With random files, the LOC function either gets the record number just read from a GET-File statement, or returns the record number just written from a PUT-File statement.

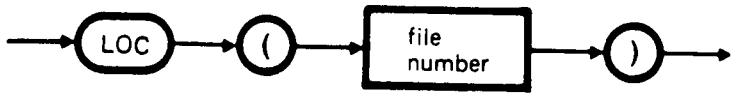


Figure 12-16 LOC Function

Where

SYNTAX ELEMENT	MEANING
file number	is a numeric expression rounded to the nearest integer. It is the number of the buffer associated with the file

Example

```
10 OPEN "R",2,"TOWNS",80
20 FIELD 2,20 AS F1$,20 AS F2$,
   20 AS F3$, 20 AS F4$
30 Y=1
    .
    .
    .
100 A$="MILAN              "
110 GET 2,Y
120 Y=Y+1
130 IF F1$=A$ THEN PRINT
```

here F1$ is a field name. If F1$ matches A$, the record number in which it was found is displayed

```
    "FOUND IN RECORD";LOC(2):
    CLOSE:END
140 GOTO 110
    .
    .
    .
```

Remark

If the file is open, but no disk I/O has been performed yet, LOC returns the value Ø.

READING RECORDS FROM A RANDOM FILE

To read records from a random file you must open it, specifying "R" as access mode. The GET-File statement allows you to read a record from a random file. GET-File specifies both the file number and the number of the record to be read. When executing a GET-File, the contents of the specified record is transferred into the file buffer.

To access a single data item stored in the buffer (field name) you may use either:

- a LET statement (if you want to assign it to a program variable), or
- a PRINT, PRINT USING, LPRINT, or LPRINT USING statement (if you want to display or print it)

Note: If you have to assign, display or print a field name to be converted to a number you must convert it using a CVI, or CVS or CVD function.

Note: The LOC function returns the number of the record just read by a GET-File or written by a PUT-File statement.

The following program steps are required to read data from a random file.

DISK FILE HANDLING

STEP	OPERATION	EXAMPLES
1	open the file, specifying "R" as access mode and (optionally) the record length	1Ø OPEN "R",#2,"1:DIR",22
2	Structure the buffer by a FIELD statement	2Ø FIELD#2,15 AS A$,5 AS B$,2 AS C$
3	Read a record from the file (variable A contains the record number).	1ØØ GET#2,A
4	extract data from the buffer by either a LET or a PRINT (PRINT USING) statement. Numeric values (stored in string format within the buffer) must be converted to numbers using the "convert" functions: CVI, CVS and CVD	1ØØ A1$=A$ 12Ø PRINT B$ 13Ø I%=CVI(C$)
5	to read another record, continue at step 3. Otherwise, go to step 6	
6	close the file (unless you want to chain a program working on the same file)	5ØØ CLOSE#2

Note: In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

GET-File (PROGRAM/IMMEDIATE)

Reads a record from a random file.

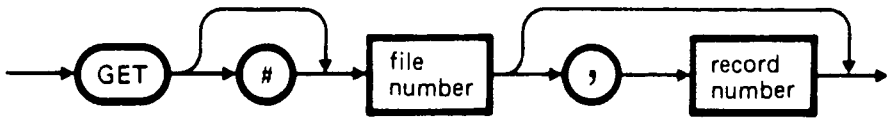


Figure 12-17 GET-File Statement

Where

SYNTAX ELEMENT	MEANING
file number	is a numeric expression, whose rounded value specifies the number of the buffer associated with the file
record number	is a numeric expression whose rounded value specifies the number of the record to be read (i.e. transferred to the buffer). If omitted, the current record is read. The smallest record number is 1, the largest 32767 Note: The current record is the record whose number is one higher than that of the last record accessed. The first time you access a random file (without specifying a record number) the current record number is set equal to 1

Examples

DISPLAY	COMMENTS
LIST 1Ø OPEN "r",1,"1:RAND",48 2Ø FIELD 1,2Ø AS R1$,2Ø AS R2$,8 AS R3$ 3Ø FOR L=1 TO 4 4Ø GET 1,L 5Ø PRINT R1$,R2$,CVD(R3$) 6Ø NEXT	This program retrieves information stored in the specified file. The data read into the buffer may be accessed by the program. This is done here by a PRINT statement (see statement 5Ø).

```
7Ø CLOSE 1
8Ø END
Ok
RUN
Super man      USA          11234621
robin hood     England      2346210l
    :
    :
Ok
```

These data items were written to the file by the PUT-File statement.

CVI/CVS/CVD

Convert string values to numeric values.
CVI converts a 2-character string to an integer
CVS converts a 4-character string to a single precision number
CVD converts a 8-character string to a double precision number

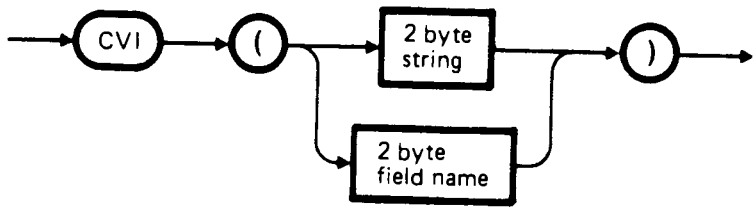


Figure 12-18 CVI Function

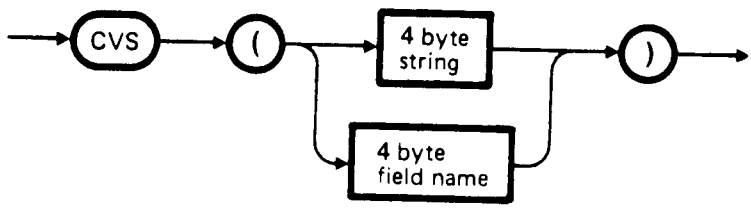


Figure 12-19 CVS Function

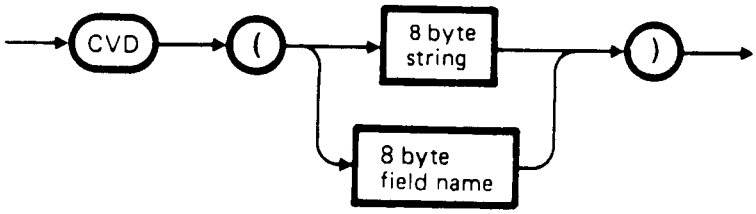


Figure 12-20 CVD Function

Examples

```
1Ø X#=CVD(N$)
2Ø Y!=CVS(R1$)
```

UPDATING RECORDS OF A RANDOM FILE

To update a random file, read in each record to be updated and rewrite it, as indicated by the following table.

STEP	OPERATION
1	open the random file
2	divide the buffer into fields
3	read the record to be updated
4	extract data items from the buffer to display them or assign them to program variables
5	insert new values into the buffer fields
6	write the updated record
7	to update another record, continue at step 3. Otherwise, go to step 8
8	close the file (unless you want to chain a program working on the same file)

Example

DISPLAY	COMMENTS
LIST 1Ø OPEN "r",1,"1:filetext",128 2Ø FIELD 1,128 AS A$ 3Ø INPUT "record number ";RNUM 4Ø GET 1,RNUM 5Ø PRINT A$ 6Ø INPUT "give me data ";PP$ 7Ø LSET A$="new data --"+PP$ 8Ø PUT 1,RNUM 9Ø INPUT "CONTINUE (y/n) ";R$ 1ØØ IF R$="y" THEN 3Ø 11Ø CLOSE Ok RUN record number ? 1 new datapoloo give me data ? gio CONTINUE (y/n) ? y record number ? 1 new data --gio give me data ? pol CONTINUE (y/n) ? n Ok	Statement 1Ø opens a random file, called filetext and residing on the diskette mounted in drive 1. Statement 2Ø specifies only one field name in this case . Statement 4Ø reads the record to be updated, whose number is entered via keyboard by statement 3Ø. Statement 5Ø displays data from the buffer. Statement 7Ø inserts new values into the buffer field, chaining the string variable PP$ to the string constant "new data __". Statement 8Ø writes the updated record. Statements 9Ø and 1ØØ allow you to continue or to stop. Statement 11Ø closes the file

13. DEBUGGING AND ERROR RECOVERY

ABOUT THIS CHAPTER

This chapter describes the statements, and some of the techniques, used for diagnosing and correcting errors (bugs).

CONTENTS

DEBUGGING AND ERROR RECOVERY

TYPES OF ERRORS

Even accomplished programmers can rarely write an error-free program at the first attempt. There are, in general, two types of errors that can be made (excluding errors made when entering a line which have already been described in Chapter 1):

- run-time errors, which halt execution and cause an error message

- logic errors, which permit complete execution, but cause incorrect or unexpected results.

The process of finding the cause of an error (often called a "bug") is termed "debugging". The M20 provides a number of features that reduce the cost and frustration of debugging.

TYPES OF ERRORS	COMMENTS
Run-time errors (i.e. errors detected by the M20 when executing a program or an immediate line)	They may be Syntax errors (when a line contains some incorrect sequence of characters) or other types of run-time errors (NEXT without FOR, RETURN without GOSUB, etc...). You can also simulate the occurrence of a BASIC error, or generate a user defined error type (to be handled by an error trap routine). See ERROR and ON ERROR GOTO statements below.
Logic errors (i.e. errors that permit complete execution, but cause incorrect or unexpected results)	These errors are the most difficult to find. To give a simple example, assume you have written a program that is supposed to print the results of 15 calculations. When the program is run, only 11 results are printed. Obviously something is wrong, but if the program is long and complex, with many branches, loops and subroutines, finding the error is not a simple task. Perhaps you have transferred control to a statement you did not intend to and some calculation is not being performed. You could have gone wrong in many ways. In such cases, the ability to trace exactly which statements are being executed - and when - would be very useful.

TRACING PROGRAM EXECUTION

A convenient method of debugging logic errors is to trace the order of statement execution in all or part of a program. The M20 provides the following two tracing commands (they may also be used as program statements):

TRON/TROFF (PROGRAM/IMMEDIATE)

TRON (TRACE ON) causes the line number of each statement executed to be listed.

TROFF (TRACE OFF) stops the line number listing initiated by TRON.

Figure 13-1 TRON Command

Figure 13-2 TROFF Command

Example

DISPLAY

```
TRON
Ok
LIST
1Ø K=1Ø
2Ø FOR J=1 TO 2
3Ø L=K+1Ø
4Ø PRINT J;K;L
5Ø K=K+1Ø
6Ø NEXT
7Ø END
Ok
RUN
[1Ø] [2Ø] [3Ø] [4Ø]  1  1Ø  2Ø
[5Ø] [6Ø] [3Ø] [4Ø]  2  2Ø  3Ø
[5Ø] [6Ø] [7Ø]
Ok
TROFF
Ok
```

COMMENTS

TRON sets the trace flag that displays each line number of the program as it is executed. The numbers appear enclosed in square brackets.

The numbers which are not enclosed in square brackets (in the example) are the output of the statement

4Ø PRINT J;K;L

The trace flag is set to off with TROFF (or when a NEW command is executed).

INTERRUPTING PROGRAM EXECUTION

A program is interrupted if:

- you press CTRL C, or

- a STOP or END statement is executed, or

- an error message is displayed.

In any of the above mentioned cases, the M20 enters Command Mode,(except in the case of a Syntax error when M20 enters Edit Mode). If you are in Command Mode, you may display program variables (by immediate PRINT or PRINT USING statements) or change their values (by immediate LET or SWAP statements. You can continue execution by entering a CONT command (except when an error is encountered, or if you modify the program).

END (PROGRAM)

Interrupts program execution, closes all data files and returns to Command Mode.

Figure 13-3 END Statement

Remarks

Although it is not essential for a program to finish with an END statement, it is useful in that it closes all open files, and it enhances readability. The END statement is also useful in enabling the program to be terminated at the end of a branch. For example:

```
25Ø IF Z>1ØØØ THEN END
```

END statements may be placed anywhere in the program to terminate execution.

Unlike the STOP statement, END does not cause a BREAK message to be displayed. The execution of an END statement always causes a return to Command Mode. You may display the values of program variables by an immediate PRINT (or PRINT USING) statement, and you may resume execution by a CONT command (but take care as all files have been closed).

STOP (PROGRAM)

Interrupts program execution and returns to Command Mode.

Figure 13-4 STOP Statement

Remarks

Like END, a STOP statement can be used anywhere in a program. When a STOP is encountered, the following message is displayed:

Break in line nnnnn

Unlike the END statement, the STOP statement does not close files.

BASIC always returns to Command Mode after a STOP is executed. Execution is resumed by issuing a CONT command (see below).

Example

DISPLAY	COMMENTS
LIST 1Ø INPUT A,B,C 2Ø X=A*B 3Ø STOP 4Ø X=X/C 5Ø PRINT X 6Ø END Ok RUN ? 4,3,6 Break in 3Ø Ok PRINT X 12 Ok CONT 2 Ok	Statement 3Ø allows you to check and observe the first value of X before the second is calculated and displayed. Although in such a simple case the STOP statement does not appear very useful, it can be very useful in larger programs: by entering a STOP at the end of a branch, for example, the program will only stop if the branch is used. It also enables you to change some variables before the program is CONTinued: a useful diagnostic test. When the program has been sufficiently tested, you have to delete all the STOPs inserted for debugging and RENUMber the program.

CONT (IMMEDIATE)

Continues program execution after either a CTRL C has been entered, or a STOP or an END statement encountered.

Execution resumes at the point where the break occurred.

Figure 13-5 CONT Command

Characteristics

IF...	THEN...
you press CTRL C after a prompt from an INPUT statement	execution continues with the reprinting of the prompt (? followed by a blank, or prompt string).
either a STOP is encountered or and END statement	intermediate values may be examined and changed using immediate statements (PRINT, PRINT USING, LET, SWAP). Execution may be resumed with CONT or an immediate GOTO, which resumes execution at a specified line number. (Entering RUN line number instead of GOTO line number will clear all program variables.) For example:

```
1Ø INPUT A,B,C
2Ø K=A^2*5.3:L=B^3/.26
3Ø STOP
4Ø M=C*K+1ØØ:PRINT M
RUN
? 1,2,3
Break in 3Ø
Ok
PRINT L
 3Ø.7692
Ok
CONT
 115.9
Ok
```

the program has been edited during the break OR ar error is issued	CONT is invalid

ERROR TESTING AND RECOVERY

Normally, when an error is encountered, BASIC handles the error by halting execution and displaying an appropriate message. In the case of a syntax error, the M20 goes into Edit Mode. In all other cases the M20 goes into Command Mode.

Often the user wants the handling of a particular error to be different from this. This is accomplished by writing his own error-handling routine.

Through use of the ON ERROR GOTO statement, error handling routines can be entered so that execution continues with the specified line after an error occurs. Only one error handling routine may be active at any given time.

Execution of an ON ERROR GOTO Ø outside an error handling routine disables error trapping.

Execution of an ON ERROR GOTO Ø inside an error handling routine specifies normal error-handling for any error which the routine does not handle.

When an error occurs and the error trapping has been enabled, execution is transferred to the specified line. Then the ERR and ERL functions could be tested and error recovery procedures could be executed. The ERR function contains the error code, the ERL function contains the line number of the line in which the error was detected.

A user-error handling routine should check for all the particular errors the user wishes to recover from, and indicate what - to do in each case. This usually involves correcting the error, and resuming execution at the statement where the error occurred, rather than returning to Command Mode.

Figure 13-7 ON ERROR GOTO Statement

ERROR (PROGRAM/IMMEDIATE)

Simulates the occurrence of a BASIC error, or generates a user defined error.

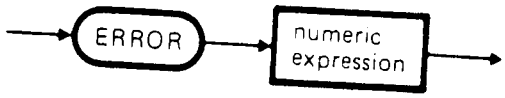


Figure 13-6 ERROR Statement

Where

SYNTAX ELEMENT	MEANING
numeric expression	the value of the numeric expression represents an error code. It must be greater than Ø and less than or equal to 255. If it is not an integer, it is rounded to the nearest integer. Note: BASIC does not use all the error codes available. The initialised error codes display the message 'Unprintable error'

Characteristics

IF...	THEN...
the value of the numeric expression equals an error code already in use by BASIC (see Appendix F)	the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be displayed.

For example:

```
LIST
10 S=10
20 T=5
30 ERROR S+T
```

```
40 END
Ok
RUN
String too long in line 30
Ok
```

Or, in immediate mode:

```
ERROR 15
String too long
Ok
```

the value of the numeric expression is greater than any used by BASIC error codes	the ERROR statement will generate a user-defined error. This user-defined error code may then be handled in the error handling routine (see ON ERROR GOTO below). Note: To define your own error, use a value that is greater than any used by BASIC error codes. (It is preferable to use the highest available values, so compatibility may be maintained if more error codes are added to BASIC).
an ERROR statement specifies a code for which no error message has been defined	BASIC responds with the message: Unprintable error

ON ERROR GOTO (PROGRAM)

Enables error handling and specifies the first line of the error handling routine. (Each BASIC program may only have one active error handling routine at any given time.)

Where

SYNTAX ELEMENT	MEANING
line number	is the first line of the error handling routine. It must be greater than Ø and less than or equal to 65529. Note: The statement ON ERROR GOTO Ø does not enable error trapping at a routine whose first line is zero, but, rather it disables error trapping. Thus, if ON ERROR GOTO Ø is within the error handling routine and that statement is reached with an error still pending, then the standard error message is displayed and Command Mode is entered.

Example

DISPLAY	COMMENTS
. . . 11Ø ON ERROR GOTO 4ØØ 12Ø INPUT "WHAT IS YOUR BET";B 13Ø IF B > 5ØØØ THEN ERROR 21Ø	If you enter a value of B greater than 5ØØØ, the message: HOUSE LIMIT IS $5ØØØ is displayed and execution resumes at 12Ø. If any other error is encountered statement 42Ø causes the standard error message to be displayed.
4ØØ IF ERR=21Ø THEN PRINT "HOUSE LIMIT IS $5ØØØ" 41Ø IF ERL=13Ø THEN RESUME 12Ø 42Ø ON ERROR GOTO Ø . . .	

Characteristics

IF...	THEN...
Error trapping has been enabled	all errors detected, including immediate mode errors, will cause a jump to the specified error handling routine.
Line number does not exist	an "Undefined line" message is displayed.
an ON ERROR GOTO Ø is executed	error trapping is disabled. Subsequent errors will display a standard error message and halt execution.
an ON ERROR GOTO Ø is executed within an error trap routine	BASIC displays the standard error message for the error which caused the trap and stops. Note: It is recommended that all error handling routines execute an ON ERROR GOTO Ø, if an error is encountered for which there is no recovery action.
an error occurs during execution of an error handling routine	the BASIC error message is displayed and execution terminates. Error trapping cannot be activated within an error handling routine.

Remark

"Overflow" and "division by zero" errors cannot be trapped.

ERL/ERR

When an error occurs the ERL function returns the line number of the line in which the error was detected, and the ERR function returns the error code.

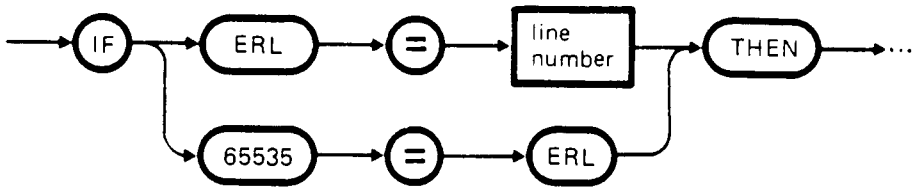


Figure 13-8 ERL Function

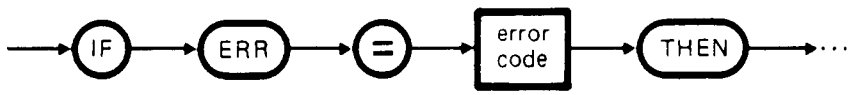


Figure 13-9 ERR Function

Characteristics

The ERL and ERR functions are usually used in IF...THEN...ELSE or IF... GOTO...ELSE statements to direct program flow in the error handling routine.

IF...	THEN...
the statement that caused the error was an immediate statement	ERL will contain 65535. To test if an error occurred in an immediate statement, use: IF 65535=ERL THEN... Otherwise, use: IF ERR = error code THEN... IF ERL = line number THEN...
the line number is not on the right side of the relational operator	it cannot be renumbered by RENUM.

Example

DISPLAY	COMMENTS
LIST 1Ø REM RECTANGLE2 2Ø ON ERROR GOTO 7Ø 3Ø INPUT "Length and Width";L,W 4Ø IF (L<Ø) OR (W<Ø) THEN ERROR 2ØØ 5Ø PRINT "Area=";L*W;" L=";L;" W=";W 6Ø GOTO 3Ø 7Ø IF (ERR=2ØØ) AND (ERL=4Ø) THEN PRINT "L or W<Ø":RESUME 3Ø 8Ø ON ERROR GOTO Ø 9Ø END Ok RUN Length and Width? -2,5 L or W<Ø Length and Width? 2,5 Area= 1Ø L= 2 W= 5 Length and Width?^C **Break in 3Ø** Ok	If you enter a negative value for L or W, the error handling routine is activated and the system displays: L or W<Ø Execution is resumed at statement 3Ø (see RESUME statement below). Note the use of ERR and ERL functions in the error handling routine.

Remarks

These functions can also be used as regular BASIC functions.

For example:

```
PRINT ERR

PRINT "Too big", ERL

I% = ERR
```

RESUME (PROGRAM)

Resumes execution after the error handling routine has been entered.

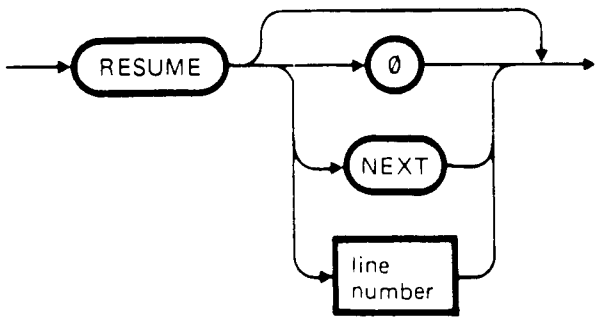


Figure 13-1[illegible] RESUME Statement

Where

SYNTAX ELEMENT	MEANING
Ø	Execution will resume at the statement which caused the error. Note: RESUME Ø and RESUME are equivalent
NEXT	Execution will resume at the first statement after the one causing the error
line number	Execution will resume at the specified line number

Remark

A RESUME statement that is not within an error handling routine causes a "RESUME without error" message to be displayed.

Examples

DISPLAY

```
LIST
1Ø REM RECTANGLE3
2Ø ON ERROR GOTO 7Ø
3Ø INPUT "Length and Width";L,W
4Ø IF (L<Ø) OR (W<Ø) THEN ERROR 2ØØ
5Ø PRINT "Area=";L*W;" L=";L;" W=";W
6Ø GOTO 3Ø
7Ø IF (ERR=2ØØ) AND (ERL=4Ø) THEN RESUME
8Ø ON ERROR GOTO Ø
9Ø END
Ok
RUN
Length and Width? -2,5
^C
Ok
```

COMMENTS

If you enter a negative value for L or W, the error handling routine is activated. In this case the routine resumes execution at the statement which caused the error, thus an endless loop is entered.
To stop execution press:

CTRL C

```
7Ø IF (ERR=2ØØ) AND (ERL=4Ø) THEN RESUME NEXT
RUN
Length and Width? -2,5
Area=-1Ø  L=-2  W= 5
Length and Width? ^C
Break in 3Ø
Ok
```

Correcting line 7Ø in this way, the error is 'ignored".

```
7Ø IF (ERR=2ØØ) AND (ERL=4Ø) THEN RESUME 3Ø
RUN
Length and Width? -2,5
Length and Width? 2,5
Area= 1Ø  L= 2  W= 5
Length and Width? ^C
Break in 3Ø
Ok
```

Correcting line 7Ø in this way, the error handling routine resumes execution at statement 3Ø.

14. GRAPHICS

ABOUT THIS CHAPTER

This chapter provides an introduction to the graphics facilities available with BASIC on the M20. On a computer, 'graphics' is the way information is conveyed in 'picture' form. This chapter explains how to execute graphics operations on the M20; the term 'graphics' covers any combination of text and geometric forms.

CONTENTS

INTRODUCTION

There are two types of screen available with the M20: one has a black and white display, the other a colour display. For both displays you can select either a 512 x 256 or a 480 x 256 pixel screen display (256 scanlines of either 512 or 480 pixels, where the term pixel is a contraction of "picture element" and scanline is a row of pixels).

In a black and white system there exists in memory one Bit Map where each bit corrisponds to a pixel (bit = 0 for black, bit = 1 for white).

A colour system may be either a 4-colour or an 8-colour system, depending on whether 4 or 8 concurrent colours are permitted.

In a 4-colour system there are two superimposed Bit Maps, where each pair of bits corresponds to a pixel. Thus, four possible colour numbers may be associated with each pixel as two bits may generate the number 0, 1, 2 and 3.

In an 8-colour system there are three superimposed Bit Maps where each three bits corresponds to a pixel. Thus, eight colour numbers may be associated with each pixel as three bits may generate the number 0 to 7.

The maximum dimensions of the video image is 225 mm by 140 mm.

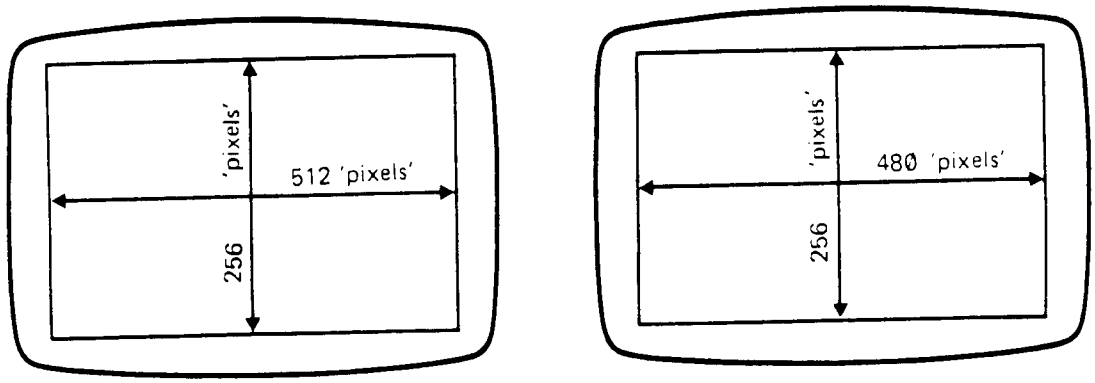


Figure 14-1 Display Modes (512 x 256 or 480 x 256)

Each line of text on the video can contain either 64 or 80 characters. The space between lines can also be varied: a screen can display from 16 lines (minimum) to 25 lines (maximum).

With the 4-colour version of the video, characters and graphics can be

displayed using four colours, selected from the eight colours provided with the system. With the 8-colour version, characters and graphics can be displayed using all the eight colours simultaneously. The eight colours are: black, green, blue, cyan, red, yellow, magenta and white.

With a colour display the background colour is normally black and the foreground colour (the colour in which characters and graphics are displayed) is green. For the black and white display the background is normally black and the foreground white. You can alter these values to suit your needs; later on in this chapter there is a description of how these values can be changed (see the COLOR statement). For black and white videos your only option is to reverse the normal method of characters or graphics display (providing black characters/graphics on a white background).

WINDOWS

You can subdivide the screen into rectangular areas, called windows. A maximum of sixteen windows can be opened (the PCOS SBASIC command may be used to preallocate memory for a specified number of windows). A window is a portion of the screen that you can work on as if it were a screen in its own right. The operations you perform within a window have no effect on any other window you may have opened. The dimensions of the windows you open are under your control by using the WINDOW statement, which is explained later in this chapter. A window can be used to display text, or graphics or both text and graphics. If you want to use graphics in a window you can either select your own set of co-ordinates or you can use the default values supplied by the system(these options are described in the section explaining the SCALE statement, later on in this chapter).

The default co-ordinate system is the hardware co-ordinate system (in pixel) only if the video has not been split into windows and the 512 x 256 display mode has been used. In any other case the default co-ordinate system is a user co-ordinate system, as the window is subdivided in 512 units along the x-axis and 256 units along the y-axis and the origin (Ø,Ø) is placed at the lower left-hand corner of the window.

If you want to use a window to display and operate on lines of text, the origin is placed at the top left-hand corner of the window, at the position (1,1) where you enter you first character. (The text co-ordinates are always expressed in terms of column and row). The CURSOR statement (described later in this chapter) allows you to move to any character position within the window.

Remember that graphics co-ordinates and text co-ordinates are totally independent. Every window has two cursors, one for graphics, another for text. The graphic cursor does not move automatically when graphic statements are executed. Both text and graphic cursors may be positioned by using the CURSOR statement.

If an attempt is made to draw a figure (or a label string, see "Professional Computer Operating System (PCOS)- User Guice") which falls outside the current window boundary (i.e. outside the window you are working on), only the portion of the figure (or the label string) which falls inside the window boundary is drawn, and the remainder is "clipped".

OPENING WINDOWS

When BASIC is initially entered, the default co-ordinate system exists, and the entire screen is one single window, window number 1. You may define a new window to be a rectangular portion of any existing window. To do that, you must use the WINDOW statement.

WINDOW - TO OPEN A WINDOW (PROGRAM/IMMEDIATE)

Opens a new window by subdividing the current window (which is called the "parent" window). The current window is the one you are working within.

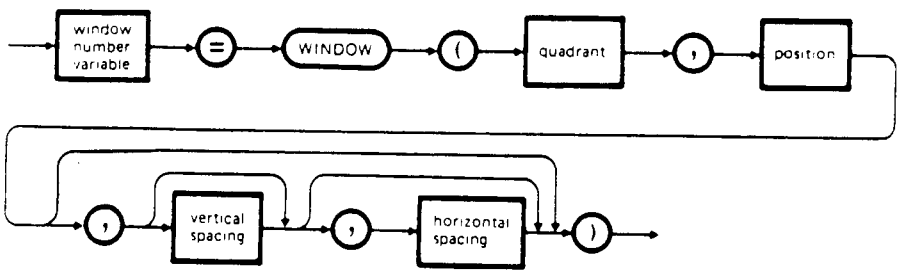


Figure 14-2 WINDOW Statement - To open a window

Where

SYNTAX ELEMENT	MEANING
window number variable	this is an integer variable, to which the system assigns an integer value which identifies the window you are opening. This value will be in the range 2 to 16. The system assigns values in ascending numeric sequence. The first window is known to the system as window number 1, the second 2, etc. Thus, if you are working within window 4 and decide to open another window, the system will assign number 5 to the new window, unless any of the windows 2 or 3 have been closed (see CLOSE WINDOW statement below), in which case the new window will be assigned the smallest available window number. Window 1 can never be closed. The window being split (in this example window 4) is called the parent window because the new window is a subdivision of it. Note: The complete screen is considered by the system to be the first window and is therefore assigned window number 1. If it is split to generate other windows, it maintains the number 1
quadrant	specifies in which part of the parent window a new window will be opened. There are four options: Ø top section of the parent window 1 bottom section 2 left-hand section 3 right-hand section
position	this parameter defines the position where the parent window is to be split to open the new window. If the value of 'quadrant' is Ø or 1 then a horizontal split will be made. The value

provided for 'position' is an integer number of scanlines within the range 1 to 255.

Note: The splitting line (which is not drawn) is always calculated from the top of the parent window.

If the value of 'quadrant' is 2 or 3 a vertical split will be made. In this case the integer provided for 'position' will be an integer number of characters within the following range:

lower limit = 1

upper limit = (width of the parent window) - 1

Note: If position = -1, then the parent window will be split in half (vertically or horizontally depending on the value of the quadrant).

If the 'quadrant' value is 2 or 3, the split is calculated from the left-hand side of the parent window.

vertical spacing

this is an optional parameter which sets the number of scanlines for each line of text, for the window being opened. The minimum value for 'vertical spacing' is 1Ø scanlines; this provides 25 lines of text on the whole screen. The maximum value is 16, providing 16 lines of text. If both the 'vertical spacing' and the horizontal spacing parameters are omitted, then the vertical spacing of the parent window is assumed. However if the vertical spacing is omitted but the horizontal spacing is given and is different than that of the parent window, then the resulting vertical spacing will be 16 if the horizontal spacing is 8, and 1Ø if the horizontal spacing is 6

horizontal spacing — this is an optional parameter which sets the space between characters in a line of text, for the window being opened. The 'horizontal spacing' parameter is expressed in terms of 'pixels' and can have one of two values 6 or 8. The first of these values gives 8Ø characters per full screen line and the second 64. If this parameter is omitted, then the horizontal spacing of the parent window is assumed

Note: When a new window is opened the previous contents of that area on the screen are cleared and the background and foreground colours of the parent window are assumed.

We have already seen that it is possible to specify the vertical and horizontal spacing of a window at the time it is opened.

In some cases it may be necessary to vary these spacing values in a window which has already been opened. You can do this using the WINDOW statement with the parameters quadrant and position set to zero.

WINDOW - TO SET WINDOW SPACING (PROGRAM/IMMEDIATE)

This window statement is handled as a special case. A new window is not opened. Instead the number of the current window is returned. This statement can be used to vary character spacing and/or line spacing values for an existing window.

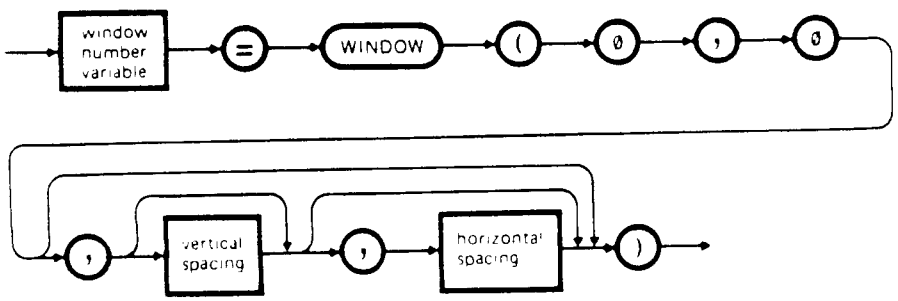


Figure 14-3 WINDOW Statement - To Set Window Spacing

Where

SYNTAX ELEMENT	MEANING
window number variable	the parameter 'window number variable' is an integer variable to which the system assigns the number of the current window
Ø	value of the 'quadrant' parameter
Ø	value of the 'position' parameter
vertical spacing	this is an optional parameter which sets the number of scanlines for each line of text, for the existing window. The minimum value for 'vertical spacing' is 1Ø scanlines; this provides 25 lines of text on the whole screen. The maximum value is 16, providing 16 lines of text. If both the 'vertical spacing' and the 'horizontal spacing' parameters are omitted, then the vertical spacing is not changed. However if the vertical spacing is omitted, but the horizontal spacing is given and is different than its current value, then the resulting vertical spacing will be 16 if the horizontal spacing is 8, and 1Ø if the horizontal spacing is 6
horizontal spacing	this is an optional parameter which sets the space between characters in a line of text, for the existing window. The 'horizontal spacing' parameter is expressed in terms of 'pixels' and can have one of two values 6 or 8. The first of these values gives 8Ø characters per full screen line and the second 64. If this parameter is omitted, then the horizontal spacing is not changed

Remark

When you switch the system on there is only one window, window number 1, consisting of the entire screen. The horizontal spacing value is 8, and the vertical spacing is 16, giving a display mode of 64 characters across by 16 text lines down. You can change the display mode to 8Ø by 25 by

using the WINDOW statement immediately after entering BASIC, and specifying the horizontal spacing as 6 pixels and the vertical spacing as 1Ø scanlines.

Examples

IF you enter..	THEN...
A=WINDOW(Ø,1ØØ) CR	the screen is split horizontally, the new window is opened in the top part of the parent window. The new window will have a height of a hundred scanlines and line spacing is given the value of the parent window
A=WINDOW(Ø,1ØØ,14) CR	as above, except the vertical line spacing is specified as 14 scanlines
B=WINDOW(2,5Ø) CR	the screen is split vertically, the new window is opened in the left-hand part of the parent window. The new window will be 5Ø character positions wide. Line-spacing has the value of the parent window
A=WINDOW(Ø,1ØØ) CR PRINT A CR	it is always possible to find out what number the operating system has assigned to an existing window If the WINDOW statement opens a window which is identified by the variable A, the operating system assigns an integer value to this variable. This can be displayed by a PRINT A statement
A=3:D=4Ø:F=15:G=8 CR W=WINDOW(A,D,F,G) CR	the screen is split vertically, the new window is opened in the right-hand part of the parent window, the vertical spacing is 15 scanlines and each full text line may contain 64 characters
W2=WINDOW(1,2Ø5,16) CR W3=WINDOW(Ø,4Ø,16) CR W4=WINDOW(2,2Ø,16) CR W5=WINDOW(3,4Ø,16) CR	these WINDOW statements split the screen into five windows. The splitting sequence is shown in the following figures

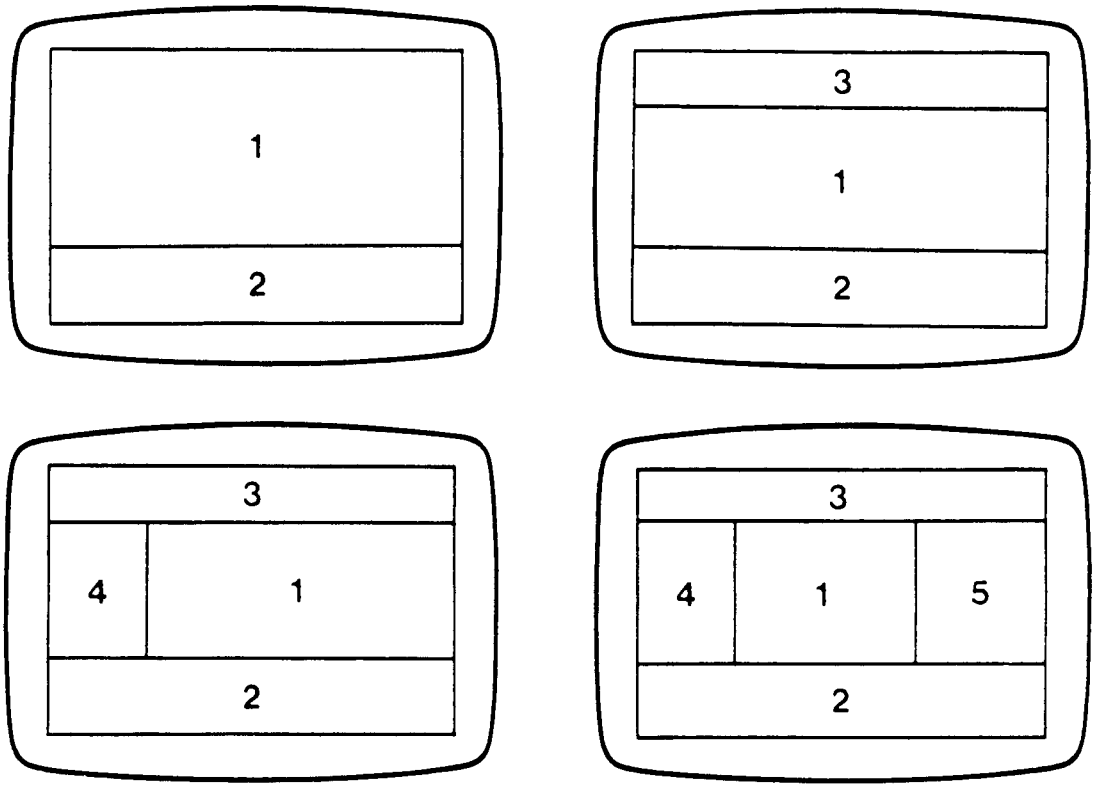


Figure 14-4 Sequence Opening Windows

USING THE WINDOWS

By using the windows, it is possible to display a number of texts or diagrams, each with a different text spacing or with different foreground and background colours. You can clear whatever appears in a window at any time by using a CLS Statement.

It is also possible to define a user co-ordinate system for every opened window, using the SCALE statement (see later in this chapter).

The WINDOW statement below allows you to select a window.

WINDOW - TO SELECT A WINDOW (PROGRAM/IMMEDIATE)

This statement selects a window. The window selected becomes the 'current window'.

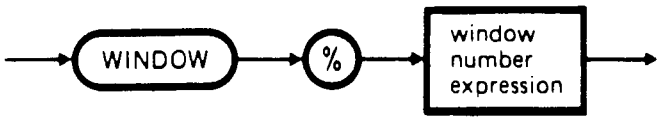


Figure 14-5 WINDOW Statement - To select a window

Where

SYNTAX ELEMENT	MEANING
window number expression	this selects the window to become the current window. It is a numeric expression whose value is rounded to the nearest integer to represent the window number. It has a value between 1 and 16 and it must correspond to an existing window, otherwise an error occurs

Examples

IF you enter...	THEN...
WINDOW %A CR	the system starts to operate within the window which was assigned the value of the variable A when it was opened. If the value assigned to the variable A is known, (e.g. 2) then this statement could be entered as follows: WINDOW %2 CR

WINDOW %1 **CR**

the system starts to operate within the window to which the operating system assigned the value 1. As you know, this is the main window (i.e. the whole screen or what is left of it).

Remarks

With the 4-colour version of the M20 you can work with 4 colours on the whole screen. These colours are chosen from a set of 8 possible colours.

Obviously, this choice does not exist with the black and white version. The COLOR - GLOBAL COLOUR SET SELECTION statement is used on the 4-colour system to choose the set of colours to be used.

COLOR - GLOBAL COLOUR SET SELECTION (PROGRAM/IMMEDIATE)

Selects 4 of the 8 colours for use on a 4-colour display.

With a black and white or an 8-colour display it may be used, but it has no effect.

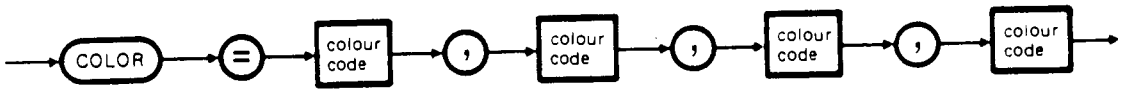


Figure 14-6 COLOR - Global Colour Set Selection Statement

Where

SYNTAX ELEMENT	MEANING
colour code	is a numeric expression having an integer value in the range Ø to 7. Each of these values corresponds to a colour as in Table 14-1. If the numeric expression value is not integer, it is rounded to the nearest integer.

Colour Codes

The following table specifies the numeric code for each colour. It is valid for both 4-colour and 8-colour systems.

COLOUR CODE	COLOUR
Ø	black
1	green
2	blue
3	cyan
4	red
5	yellow
6	magenta
7	white

Table 14-1

Colour Numbers

In many graphics statements, an optional parameter is a "colour number". In a 4-colour system the colour number is an integer from Ø to 3 corresponding to the order position of the four colour codes given in the COLOR-Global Colour Set Selection statement. This is not to be confused with the colour code shown in the table above. In a 4-colour system, if this COLOR statement is not executed, the four default colours are: black, green, blue and red. Note that this is as though the statement COLOUR=Ø,1,2,4 has been executed.

The COLOR statement described above has no effect either when used on a black and white system, because the colour numbers are defined as Ø for black and 1 for white, or when used on an 8-colour system, because there is no distinction between colour codes and colour numbers, and, there are no default colours (as all the colours are present).

We have already seen that each window has a foreground colour and a background colour. For both the black and white and the colour systems the default value is colour number Ø for the background colour and colour number 1 for the foreground colour. For a 4-colour system the colour of the (text and graphic) cursor is either the last colour explicitly declared by the COLOR statement or red if no COLOR statement is used. For an 8-colour system the colour of the (text and graphic) cursor is always white. It is possible to change these default foreground and background colours by using a different form of the COLOR statement. This is described below.

COLOR (PROGRAM/IMMEDIATE)

Selects the background and foreground colours for a particular window.

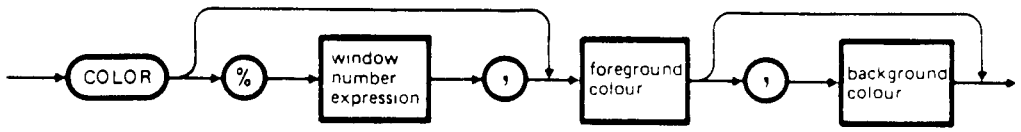


Figure 14-7 COLOR Statement

Where

SYNTAX ELEMENT	MEANING
window number expression	specifies the window to be operated on; it is optional and, if omitted, the statement operates on the current window
foreground colour	specifies the foreground colour number of the window. It may be a numeric expression whose value is rounded to the nearest integer
background colour	this is optional and is used to specify the background colour number. It may be a numeric expression whose value is rounded to the nearest integer. If omitted, the previously specified background colour remains.

Remarks

Once the COLOR Statement has been executed, the foreground and background colours are changed accordingly. You may do one of the following to realize the change of colour requested.

1. Execute the CLS statement (described later in this chapter), to supply the window with a new background colour.

2. Execute the PRESET statement (described later in this chapter), to colour parts of the window with a new background colour.

3. Display a text to change the foreground and/or background colour for that part of the window where the new text appears.

Examples

IF you enter...	THEN...
COLOR Ø,1 CR	the current window will have a white background and a black foreground
COLOR %A,Ø,1 CR	as above, but the statement operates on the window identified by the variable A

Remarks

If the user enters COLOR Ø.1 instead of COLOR Ø,1, further character input from keyboard will be invisible (as Ø.1 is rounded to Ø). To recover, enter CLEAR or the COLOR statement in the correct way.

CLS (PROGRAM/IMMEDIATE)

Clears the contents of either the current window or a specified window. To clear a window means to fill it with its background colour.

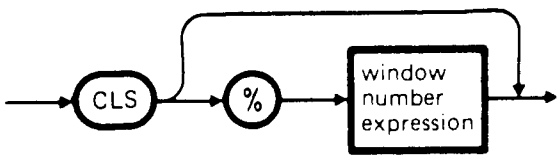


Figure 14-8 CLS Statement

Where

SYNTAX ELEMENT	MEANING
window number expression	this selects the window to be operated on. It is a numeric integer expression which represents a window number. The use of this parameter is optional. If it is not specified, the operation is performed on the current window

SCALE (PROGRAM/IMMEDIATE)

Allows you to change to any user co-ordinate system, defining a scale between the default co-ordinates and the user co-ordinates.

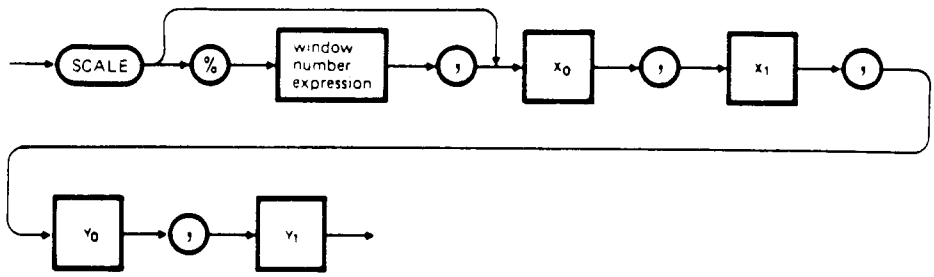


Figure 14-9 SCALE Statement

Where

SYNTAX ELEMENT	MEANING
window number expression	a numeric integer expression selecting the window. If omitted the current window is selected

x$\emptyset$,x1,y$\emptyset$,y1

window dimensions (user coordinates):

x$\emptyset$: left-hand side of the window (i.e. x minimum)

x1: right-hand side of the window (i.e. x maximum)

y$\emptyset$: bottom of the window (i.e. y minimum)

y1: top of the window (i.e. y maximum)

Note: x1 - x$\emptyset$, y1 - y$\emptyset$ can be either positive or negative, but must never be equal to zero.

Remarks

When a SCALE statement has been executed, you must express co-ordinate values that refer to the user co-ordinate system.

The co-ordinate system is the default one if:

- no SCALE statement has been executed, or

- the statement:

  SCALE $\emptyset$, 511, $\emptyset$, 255 has been executed

Examples (4-colour display)

IF you enter...	THEN...
COLOR = 3,$\emptyset$,1,5 **CR** CLS **CR** LINE($\emptyset$,$\emptyset$)-(511,255) **CR**	The LINE statement (described later) draws a black line on a cyan background from a point specified by co-ordinates ($\emptyset$,$\emptyset$) to (511,255).

This line is shown in the figure 14-1Ø.

If no SCALE statement has been executed previously, the default co-ordinate system is adopted.

```
SCALE -1ØØØ,1ØØØ,-1ØØØ,1ØØØ CR
LINE (Ø,Ø)-(511,255) CR
```

a user co-ordinate system is adopted using a SCALE statement. Thus the same LINE statement as above displays a different image. (See the figure 14-11).

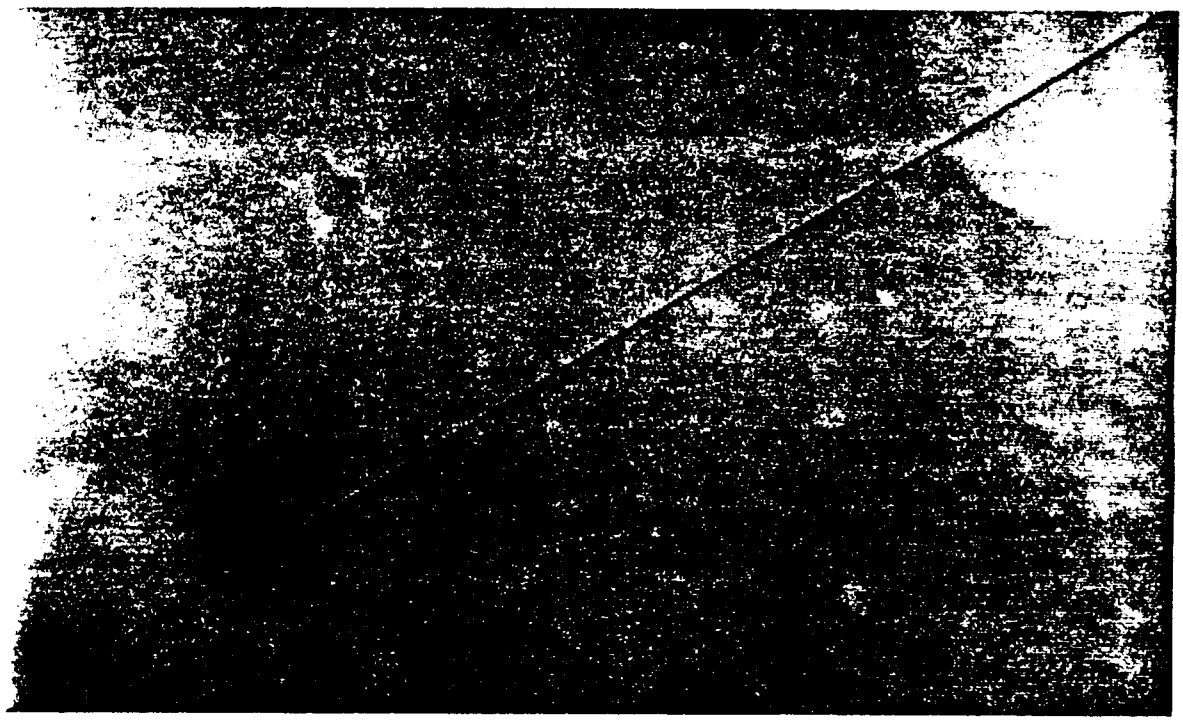


Figure 14-1Ø LINE Statement

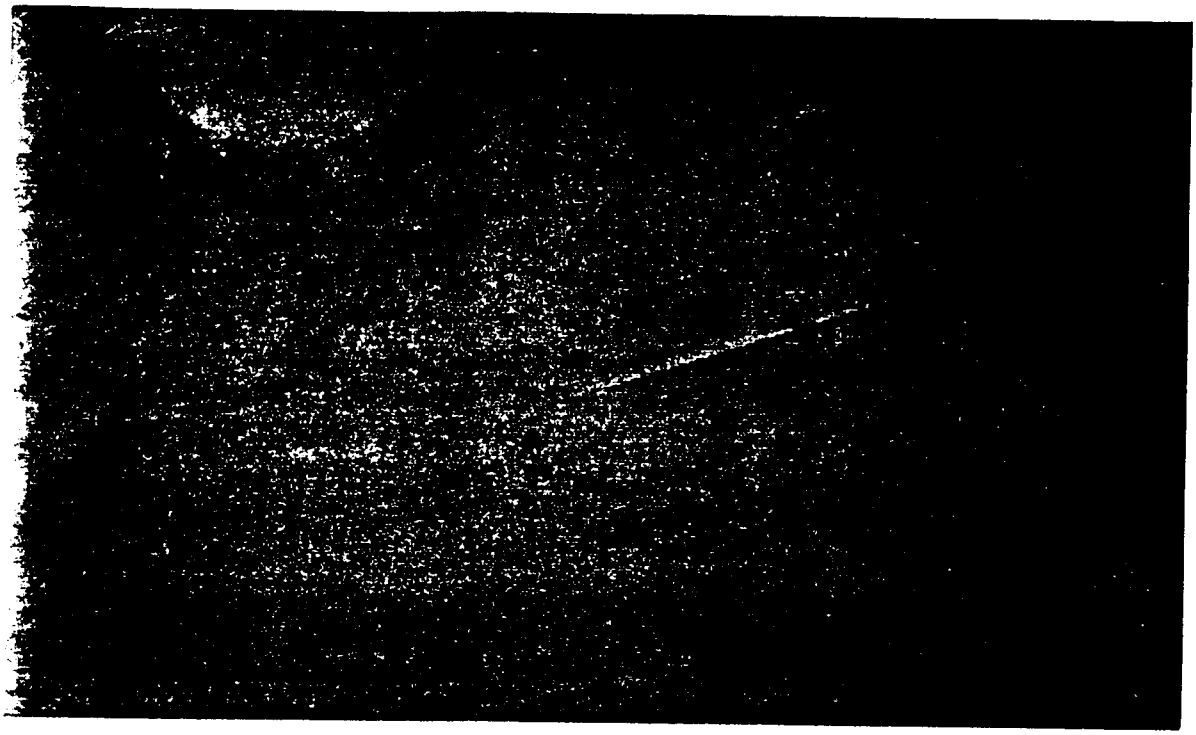


Figure 14-11 SCALE and LINE Statements

Remarks

Having defined a new co-ordinate system, using the SCALE statement, it remains in effect until a new SCALE statement is executed or you leave the BASIC environment. To find the pixel co-ordinates of a point, you must use SCALEX function and/or SCALEY function. For example the PCOS command LABEL (callable from BASIC by a CALL or EXEC statement) requires the expression of the x-pos, and y-pos parameters in the pixel co-ordinate system. Thus you have to use SCALEX and SCALEY functions if you are working with a user co-ordinate system.

SCALEX

Converts a user co-ordinate into the associated pixel co-ordinate on the x-axis of the current window.

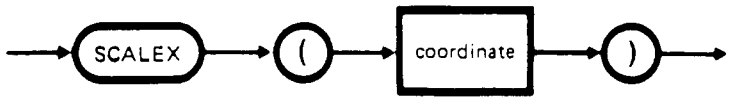


Figure 14-12 SCALEX Function

Where

SYNTAX ELEMENT	MEANING
coordinate	a user co-ordinate on the x-axis

SCALEY

Converts a user co-ordinate into the associated pixel co-ordinate on the y-axis of the current window.

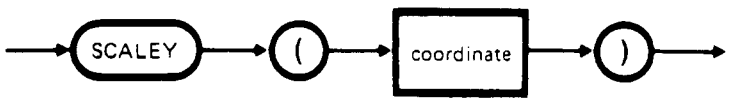


Figure 14-13 SCALEY Function

Where

SYNTAX ELEMENT	MEANING
co-ordinate	a user co-ordinate on the y-axis

CLOSING WINDOWS

Once the screen has been divided into windows, it is possible to close any of the windows, thus enlarging the size of its parent window. It is also possible to return to the "initial system" state where there is only one window. To do either of these, the CLOSE WINDOW statement is used.

CLOSE WINDOW (PROGRAM/IMMEDIATE)

Closes a selected window or all opened windows.

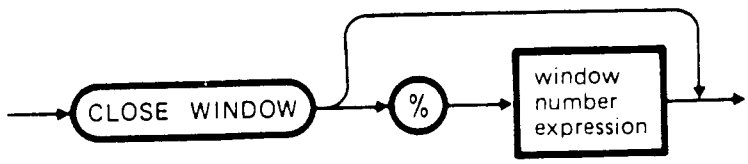


Figure 14-14 CLOSE WINDOW Statement

Where

SYNTAX ELEMENT	MEANING
window number expression	this is a numeric integer expression representing a window number. It identifies the window to be closed. If omitted, all opened windows, except the main window (i.e. the window number 1) are closed

The CLOSE WINDOW statement with the window number expression parameter closes the window identified by the parameter. The area of this window is assigned to the rectangle which was originally split to open it. The area of the window which has been closed is displayed with the background colour of the window to which the released space is assigned.

Note: The CLOSE WINDOW statement has no effect on the main window. This window can never be closed.

DISPLAYING CURSORS

Each window has two cursor positions: one for text and one for graphics. The text cursor position indicates the position where the next alpha-numeric character will be displayed. This position is expressed in terms of the text row number and the text column number.

The POS function allows you to know the position of the text cursor in the current window.

Another visible cursor may be associated with any position you desire. This cursor is called the graphic cursor, although it need not be associated with graphics, nor does it move automatically when graphic statements are executed.

By using the CURSOR statement described below, you can specify whether you want to display one of the cursors, whether to make it blink, and whether to change its shape from the default shape.

The default shape of the graphic cursor is a rectangle of 2 x 2 pixels. The default shape of the text cursor is an underbar. If you want to display one of the cursors, you can only do this in the window where you are operating; in fact as soon as you select another window, the cursor in the previous window disappears, but it is stored and appears again with the same characteristics whenever you return to that window. Bear in mind that when the text cursor is turned on, the graphic cursor is automatically disabled and viceversa; thus the two cursors cannot be displayed at the same time.

CURSOR (PROGRAM/IMMEDIATE)

There are two basic formats for this statement: CURSOR and CURSOR POINT, allowing the position and attributes of the text cursor and graphic cursor, respectively, to be specified.

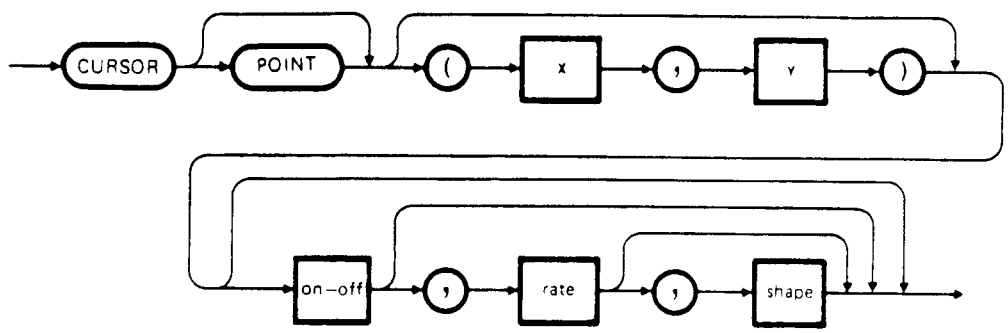


Figure 14-15 CURSOR Statement

Where

SYNTAX ELEMENT	MEANING
POINT	this is an optional keyword. It is used to operate on the graphic cursor. If omitted, operations are performed on the text cursor -
x,y	these specify where the cursor is to be placed. If we are dealing with the text cursor, then x and y represent the column and row of text respectively. If we are dealing with the graphic cursor, then x and y represent the co-ordinates of the lower left hand corner of the cursor bitmap.
on-off	specifies whether or not the cursor is to be displayed: Ø = not displayed 1 = displayed

rate

specifies whether or not the cursor is to blink:

Ø = no blinking

1-2Ø = number of blinks per second

shape

this is an optional parameter. It alters the shape of the cursor. It is the first element of a six element integer array. The array must be defined by the user; its components are the desired bit-map of the cursor. Each bit of the cursor bit-map represents a pixel.

The contents of the cursor bitmap get XORed with the contents of that part of the screen bit-map representing the screen area occupied by the cursor.

For both the text cursor and the graphic cursor the bit-map is 8 pixels wide and 12 pixels high

Examples

IF you enter...	THEN...
CURSOR POINT(8Ø,3Ø): A$=INPUT$(1) CR	the graphic cursor is positioned at the point with co-ordinates (8Ø,3Ø). The statement A$=INPUT$(1) has been entered to allow the cursor to remain in the specified position until you enter a character from keyboard

CURSOR POINT(5Ø,5Ø)1: A$=INPUT$(1) CR	the graphic cursor is positioned at the point with co-ordinates (5Ø,5Ø) and is displayed
CURSOR POINT(5Ø,5Ø)1,1: A$=INPUT$(1) CR	the graphic cursor is positioned at the point with co-ordinates (5Ø,-5Ø), it is displayed and blinks at a rate of 1 blink per second
CURSOR (32,8)1:A$=INPUT$(1) CR	the text cursor is positioned at column 32 of row 8; it is displayed and is not blinking.
CURSOR (32,8)1,Ø,A%(1): A$=INPUT$(1) CR	as above but the cursor shape has been defined by the user as an up arrow (see table below)

BIT MAP	ELEMENT	DECIMAL	HEXADECIMAL
ØØØ1ØØØØ ØØ111ØØØ	A%(1)	4152	&H1Ø38
Ø1111ØØ 11111111Ø	A%(2)	31998	&H7CFE
ØØ111ØØØ ØØ111ØØØ	A%(3)	14392	&H3838
ØØ111ØØØ ØØ111ØØØ	A%(4)	14392	&H3838
ØØ111ØØØ ØØ111ØØØ	A%(5)	14392	&H3838
ØØ111ØØØ ØØ111ØØØ	A%(6)	14392	&H3838

Table 14-2 Cursor Bit Map

Note: Remember that each element of the integer array is a sixteen bit representation.

POS (PROGRAM/IMMEDIATE)

Returns the position of the text cursor in the current window.

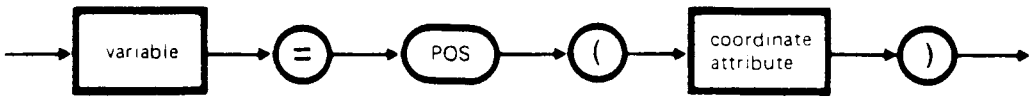


Figure 14-16 POS Statement

Where

SYNTAX ELEMENT	MEANING
variable	a numeric variable to which the system assigns an integer value. This represents either the row or column position of the text cursor within the current window (see the co-ordinate attribute below)
coordinate attribute	specifies either the row or column position. It is Ø for a column position or any non zero value for the row position

DRAWING LINES, RECTANGLES, AND CIRCLES

The M20 BASIC graphics extensions include statements allowing you to draw geometric figures.

LINE (PROGRAM/IMMEDIATE)

Draws either a line or a rectangle, or a filled rectangle, in a specified colour, with a specified diagonal.

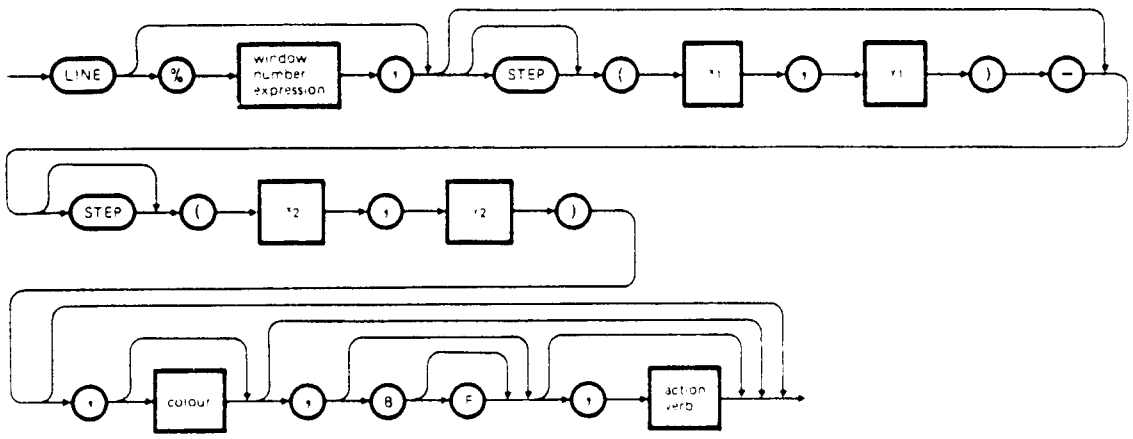


Figure 14-17 LINE Statement

Where

SYNTAX ELEMENT	MEANING
window number expression	A numeric integer expression specifying the window on which the LINE statement is to work. By default the LINE statement operates on the current window.
STEP	optional keyword. This allows the use of relative co-ordinates. Relative starting co-ordinates (x_1,y_1) are relative to the co-ordinates of the last point drawn or (in the absence of such a point) to the co-ordinates of the bottom left-hand corner of the window. Relative ending co-ordinates (x_2,y_2) are relative to the start of the line (or rectangle).
x_1,y_1	These are the co-ordinates of the starting point of the line. If omitted, the line specified by the LINE statement starts from the last point drawn, or from the bottom left-hand corner of the window, if no point has yet been drawn.
x_2,y_2	These are the co-ordinates of the end point of the line.

colour

A colour number specifying the colour with which the line or rectangle will be drawn. The default value is the foreground colour of the current window.

B (Box)

An optional parameter which allows you to trace a rectangle with its sides parallel to the edges of the window. Its diagonal is specified by the co-ordinates (x_1,y_1) and (x_2,y_2).

F (Filled)

An optional parameter which can only be used if B is also used. 'BF' draws a rectangle and fills it in with the colour specified by the colour parameter or with the foreground colour, if the colour parameter is not given.

action verb

This is an optional parameter which can assume the following values: AND, XOR, OR, NOT, PSET, PRESET.

The verb PSET indicates that the line, rectangle or filled rectangle is to be drawn in the specified colour. The verbs AND, OR, and XOR indicate that the colour of the line, rectangle, or filled rectangle is the result of a logical operation between the specified colour and the existing colour of each pixel on the screen covered by the figure. The verb NOT indicates that the colour of the line, rectangle, or filled rectangle will be the complement of the existing colour of each pixel covered by the figure. The verb PRESET indicates that the line, rectangle, or filled rectangle will be drawn in the background colour.

The default action verb is PSET.

Example (4-colour display)

DISPLAY	COMMENTS
1Ø COLOR = 4,2,4,5 2Ø CLS 3Ø LINE (2Ø6,1ØØ)-(3Ø6,1ØØ) 4Ø LINE (256,2ØØ) 5Ø LINE STEP (-5Ø,-1ØØ) 6Ø PAINT (256,15Ø)	This program draws an isosceles triangle and paints it blue (the foreground colour). The background colour is red (see figure 14-18) The PAINT statement is described below.

Figure 14-18 Drawing a Triangle

Remark

If the parameters specified for drawing a line or a box are such that a portion of the line or the box falls outside the window boundary, the line or box will still be drawn with the portion outside the window boundary clipped.

CIRCLE (PROGRAM/IMMEDIATE)

Draws a circle.

The centre of the circle is specified by the x,y co-ordinates; the radius is specified by the "r" parameter.

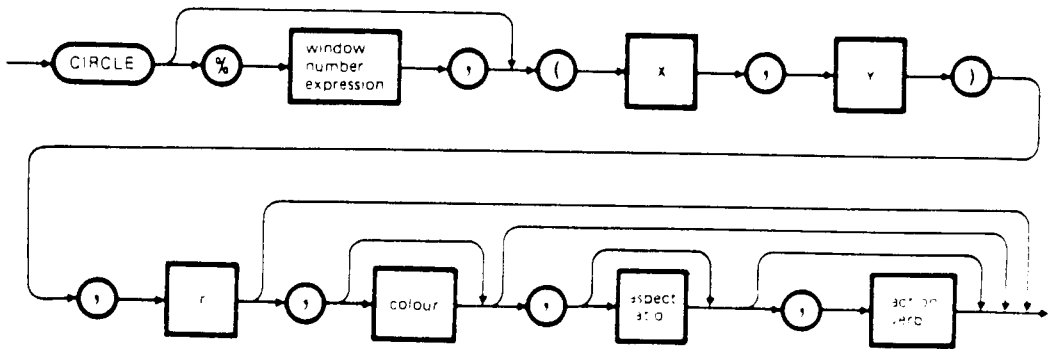


Figure 14-19 CIRCLE Statement

Where

SYNTAX ELEMENT	MEANING
window number expression	A numeric integer expression which selects the window in which the CIRCLE statement is to operate. This is optional and, if omitted, the current window is selected.
x,y	The centre of the circle
r	The radius of the circle.
colour	A colour number specifying the colour with which the circumference will be drawn. The default is the current foreground colour of the selected window.

aspect ratio

Due to the non-uniform physical distribution of the pixels on the screen, the user may specify a value of the aspect ratio to draw a true circle with different monitors.

The default value of aspect ratio (which must be a positive real number) is Ø.8Ø7. This value produces a circle, with the M20 standard monitor.

action verb

An optional parameter which may assume one of the following values: AND, XOR, OR, NOT, PSET, PRESET.

Each defines the operation which will be done for every pixel along the curve.

The verb PSET indicates that the circle is to be drawn in the specified colour. The verbs AND, OR, and XOR indicate that the colour of the circle is the result of a logical operation between the specified colour and the existing colour of each pixel covered by the curve. The verb NOT indicates that the colour of the circle will be the complement of the existing colour of each pixel along that curve. The verb PRESET indicates that the circle will be drawn in the background colour. The default action verb is PSET.

Example (4-colour display)

DISPLAY	COMMENTS
1Ø COLOR = 2,4,5,Ø 2Ø CLS 3Ø CIRCLE (1ØØ,12Ø),9Ø 4Ø CIRCLE (15Ø,13Ø),12Ø 5Ø CIRCLE (25Ø,12Ø),1ØØ 6Ø PAINT (18Ø,12Ø)	The program draws three intersecting circles. The background colour is blue, the circumferences are red (the foreground colour) and the area of intersection is also red. See figure 14-20. The PAINT statement is described below.

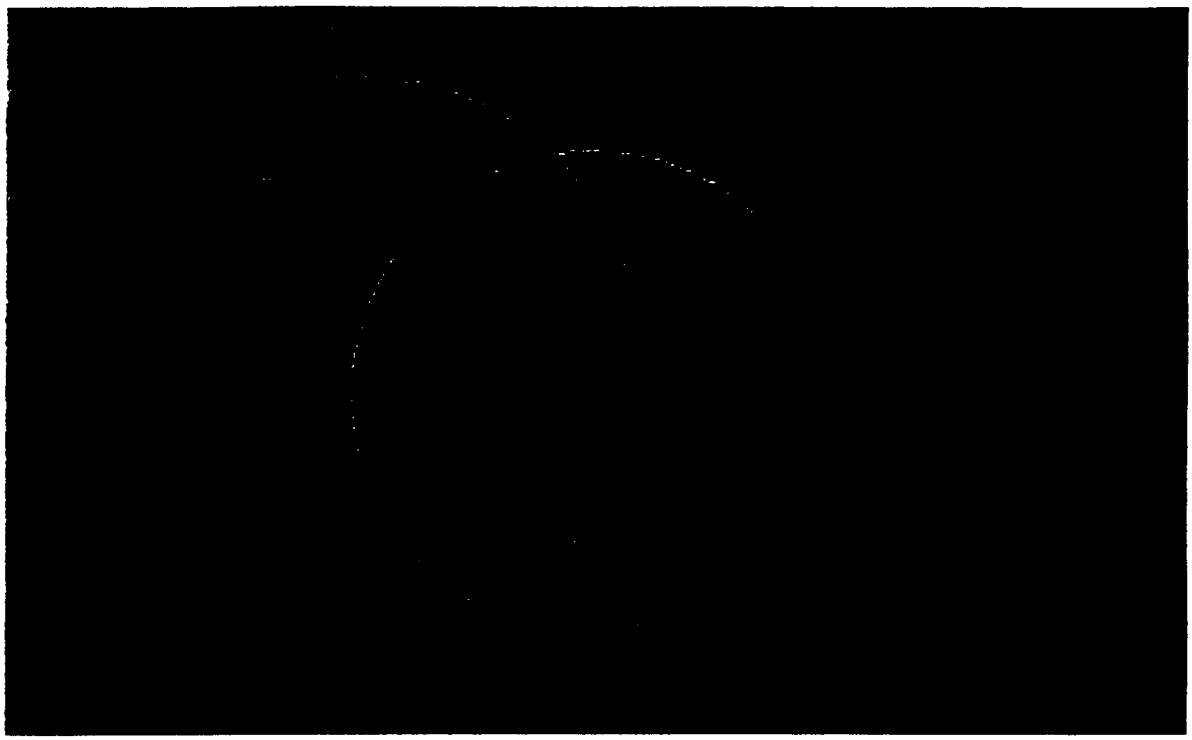


Figure 14-20 Intersecting Circles

Remarks

When a SCALE statement is used, the aspect ratio parameter is not affected by the scaling, and the radius of the circle is determined only by the horizontal scaling of the window in which the circle is to be drawn.

DISPLAYING POINTS AND PAINTING FIGURES

The most elementary graphic function is that of illuminating the position of a single point in a specified colour. This can be done using the PSET and PRESET statements.

The PAINT statement allows you to colour the area inside a closed figure.

The POINT function allows you to know the colour number of a specified pixel.

PSET (PROGRAM/IMMEDIATE)

Colours the pixel either at the specified (x,y) co-ordinates or, if the window has been scaled, the pixel nearest the (x,y) co-ordinates. It colours this pixel with either a specified or foreground colour.

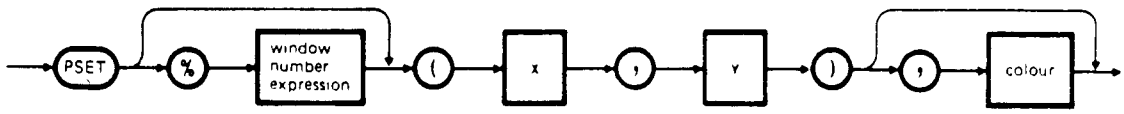


Figure 14-21 PSET Statement

Where

SYNTAX ELEMENT	MEANING
window number expression	a numeric integer expression, which represents the window in which PSET is to work. It is optional, the default is the current window
x,y	the co-ordinates used by PSET. If the x,y co-ordinates specify a point outside the window, the point will not be displayed because of the "clipping".
colour	defines the colour number for the point displayed. This parameter is optional; by default the foreground colour of the specified window is used

PRESET (PROGRAM/IMMEDIATE)

Colours the pixel either at the (x,y) co-ordinates or, if the window has been scaled, the pixel nearest the (x,y) co-ordinates. it colours this pixel with the current background colour of either the current or selected window.

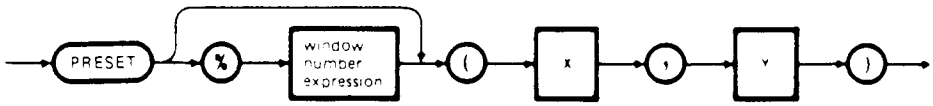


Figure 14-22 PRESET Statement

Where

SYNTAX ELEMENT	MEANING
window number expression	a numeric integer expression representing the window in which the PRESET statement is to operate. This is an optional parameter: the default is the current window
x,y	co-ordinates on which PRESET works. If the x,y specify a point outside the window, the point will not be displayed because of the "clipping"

PAINT (PROGRAM/IMMEDIATE)

Colours the area inside a closed figure, starting from tne pixel either at the specified (x,y) co-ordinates or, if the window has been scaled, the pixel nearest the (x,y) co-ordinates.

The area can be the whole or part of the specified window.

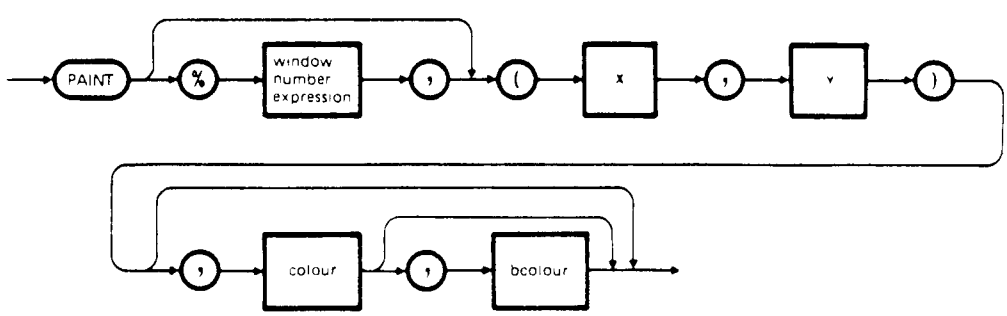


Figure 14-23 PAINT Statement

Where

SYNTAX ELEMENT	MEANING
window number expression	numeric integer expression which represents the window on which the PAINT statement is to work. This is an optional parameter. By default the current window is selected
x,y	co-ordinates of the pixel from which painting begins
colour	a colour number specifying the colour to be used to paint the window or a closed figure within it. Its default value is the current foreground colour
bcolour	a colour number specifying the colour of the border of the closed figure to be PAINTed. By default, the foreground colour of the specified window is assumed

Note: To PAINT inside a predefined closed figure, ensure that x,y fall within the border of the figure. If they fall outside the border of the figure, only the portion of the window which is outside the figure will be coloured.

Example (4-colour display)

DISPLAY	COMMENTS
1Ø COLOR = 2,5,4,Ø 2Ø CLS 3Ø CIRCLE (256,128),13Ø,2 4Ø PAINT (256,128),1,2 5Ø LINE (251,123)-STEP (1Ø,1Ø),3,BF	statement 1Ø selects four among the eight available colours. Statement 2Ø clears the screen with the background colour (in this case blue). Statement 3Ø draws a red circumference with a radius of 13Ø whose centre is (256,128). Statement 4Ø paints the circle yellow. Statement 5Ø draws a black filled in box in the middle of the circle (see figure 14-24)

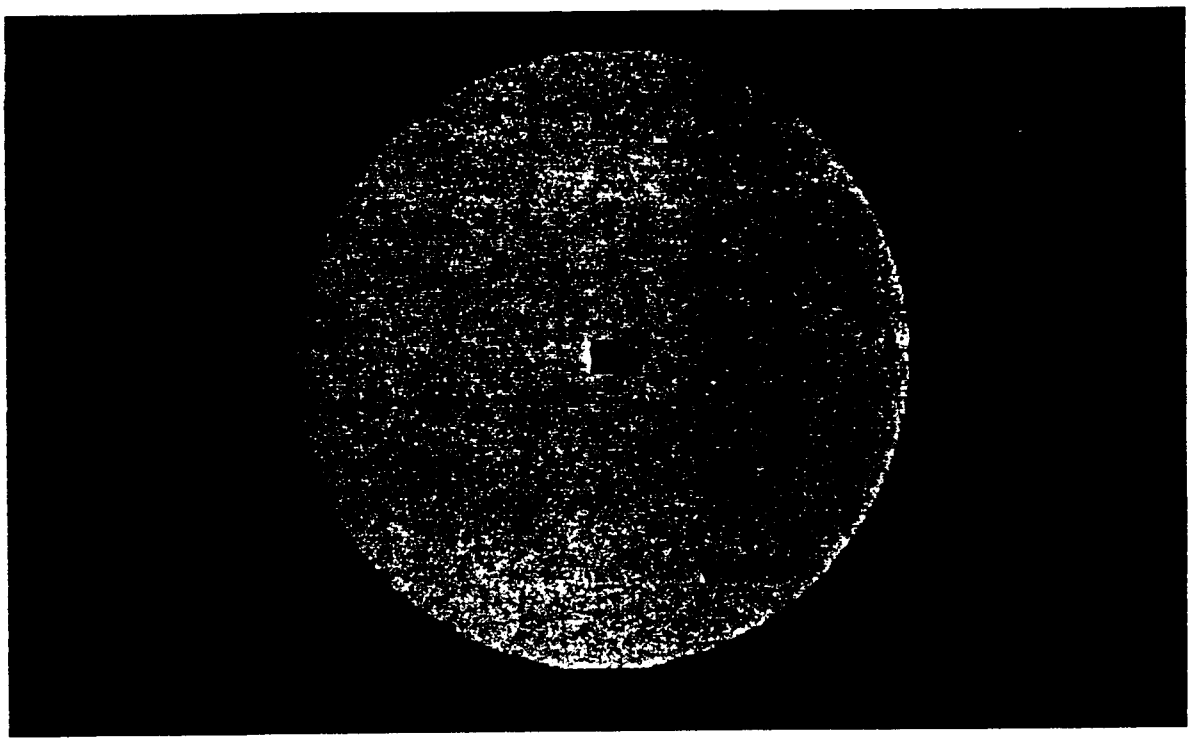


Figure 14-24 Drawing and Painting a Circle

Example (8-colour display)

DISPLAY	COMMENTS
1Ø COLOR 5,2	Statement 1Ø specifies the foreground colour (yellow) and the background colour (blue).
2Ø CLS	Statement 2Ø clears the screen with the background colour (in this case blue).
3Ø CIRCLE(256,128)13Ø,4	Statement 3Ø draws a red circumference with a radius of 13Ø whose centre is (256,128).
4Ø PAINT(256,128),5,4	Statement 4Ø paints the circle yellow.
5Ø LINE(251,123)-(1Ø,1Ø),Ø,BF	Statement 5Ø draws a black filled in box in the middle of the circle (see figure 14-24).

POINT (PROGRAM/IMMEDIATE)

Returns the colour number of the pixel either at the specified (x,y) co-ordinates or, if the window has been scaled, the pixel nearest the (x,y) co-ordinates within the current window.

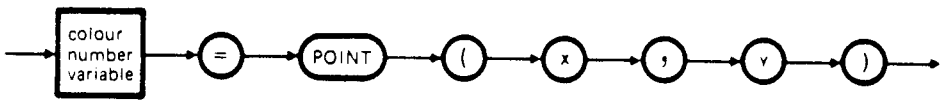


Figure 14-25 POINT Statement

Where

SYNTAX ELEMENT	MEANING
colour number variable	a variable to which the system assigns an integer value: Ø or 1 for a black and white system; in the range (Ø-3) for a 4-colour system; in the range (Ø-7) for an 8-colour system. This variable specifies the colour number of the pixel either at the (x,y) co-ordinates or, if the window has been scaled, the pixel nearest the (x,y) co-ordinates
x,y	the co-ordinates of the pixel in question

Examples

DISPLAY	COMMENTS
1Ø CIRCLE(5Ø,5Ø),2Ø	draws a circle on the screen with its centre at 5Ø,5Ø and radius of 2Ø.
2Ø PSET(5Ø,5Ø)	illuminates the pixel either at (5Ø,5Ø) with the foreground colour or, if the window has been scaled, the pixel nearest (5Ø,5Ø)

3Ø A%=POINT(5Ø,5Ø)	assigns the colour number (of the pixel either at the (5Ø,5Ø) co-ordinates or, if the window has been scaled, the pixel nearest the (5Ø,5Ø) co-ordinates) to the A% variable
4Ø PRINT A%	displays the contents of A%

SPECIAL STATEMENTS

There are three special statements: GET, PUT and DRAW.

GET and PUT

You can store the whole window or any rectangle within a window, in a one-dimensional integer array using the GET statement, or conversely you can restore anywhere on the screen a rectangle taken from a one-dimensional integer array by a PUT statement.

DRAW

The screen may be thought of as a sheet of paper on which you can draw with a pen (known as the "virtual pen"). You can move the pen to any position of the screen, drawing ("pen down") or not ("pen up").

You can move the virtual pen within a window and draw lines in a given colour using a DRAW statement.

GET - Graphics (PROGRAM/IMMEDIATE)

Stores the whole or any rectangle within a window in a specified one-dimensional integer array.

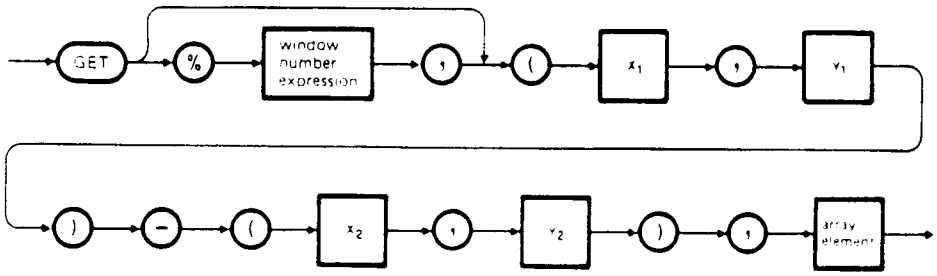


Figure 14-26 GET - Graphics Statement

Where

SYNTAX ELEMENT	MEANING
window number expression	A numeric integer expression specifying the window in which GET is to operate. The default is the current window.
x_1,y_1 x_2,y_2	Define the rectangle to be stored, the rectangle whose diagonal is specified by the line (x_1,y_1) to (x_2,y_2).
array element	The first element of the one-dimensional array which is to contain the information acquired by the GET operation. The system will fill the array as follows: the first three elements of the array will contain the width of the rectangle, the height of the rectangle, and the colour/monochrome flag, respectively. The remainder of the array will contain the bit image of each scanline of the rectangle itself. Each array element contains a string of 16 bits. This one dimensional array must have been previously dimensioned by a DIM statement. The following formula shows how to calculate the number of elements of the array: $\left(\left(\left\lceil \frac{\text{width}}{16} \right\rceil \times \text{height} \right) \times \text{DT} \right) + 3$

where:

DT=1 with a black and white display
DT=2 with a 4-colour display
DT=3 with an 8-colour display
⌈ ⌉ means take integer (always round up)

PUT - Graphics (PROGRAM/IMMEDIATE)

Displays an image previously stored in a one-dimensional integer array using a GET statement.

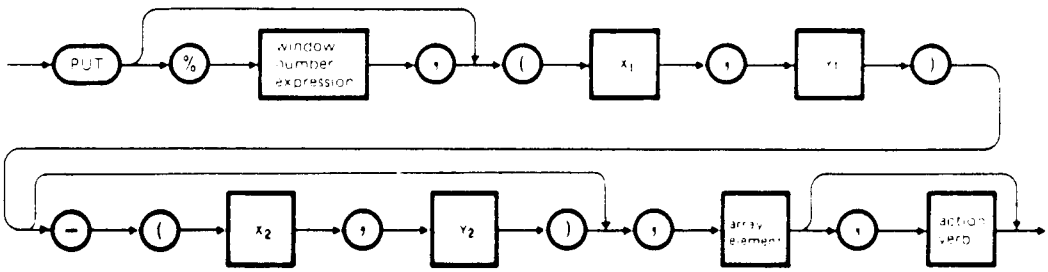


Figure 14-27 PUT - Graphics Statement

Where

SYNTAX ELEMENT	MEANING
window number expression	A numeric integer expression specifying the window in which PUT is to operate. The default is the current window.
x_1,y_1 x_2,y_2	Define the position of the rectangle to be displayed, the rectangle whose diagonal is defined by the line (x_1,y_1) to (x_2,y_2).

If this rectangle is a different size from the one in the stored array, the smaller of the two is used. If x_2 and y_2 are omitted the stored rectangle will be displayed starting from the top left-hand corner x_1, y_1.

array element

The first element of the one-dimensional array which contains the information stored by a GET operation.

action verb

An optional parameter which may assume one of the following values: AND, XOR, OR, NOT, PSET, or PRESET. Each defines the operation which will be done for every pixel within the rectangle.

The verb PSET indicates that the rectangle is to be restored directly from the stored array. The verbs AND, OR, and XOR indicate that the rectangle displayed is the result of a logical operation between the colour number of each pixel in the stored array and the existing colour number of each pixel on the screen within that rectangle. The verb NOT indicates that the existing colour number of each pixel on the screen will be complemented within that rectangle, without regard to the stored array. The verb PRESET indicates that the complement of the stored array will be displayed on the screen.

The default action verb is PSET.

Example (4-colour display)

DISPLAY	COMMENTS
1 COLOR = 2,4,5,Ø 5 DIM B%(2ØØØ) 1Ø CLS:CIRCLE (256,128),8Ø,3 2Ø LINE (19Ø,6Ø)-(35Ø,195),,BF,XOR 3Ø GET (19Ø,6Ø)-(36Ø,128),B%(Ø) 5Ø CLS:PUT (25Ø,22Ø),B%(Ø)	Statement 5 defines the array to hold the bit image. Line 1Ø clears the screen, (background is blue) and draws a black circumference.

Statement 2Ø draws a rectangle superimposed on the circle and it is filled-in in red. The XOR operation between Ø (blue), the background colour number, and 1 (red), the foreground colour number, is 1 (red). The portion of the circumference within the rectangle is coloured yellow, as the XOR operation between 3 and 1 is 2. (See figure 14-28.)

Statement 3Ø saves a section of the screen in the array B%.

Line 5Ø clears the screen and restores the saved section of the screen (see figure 14-29).

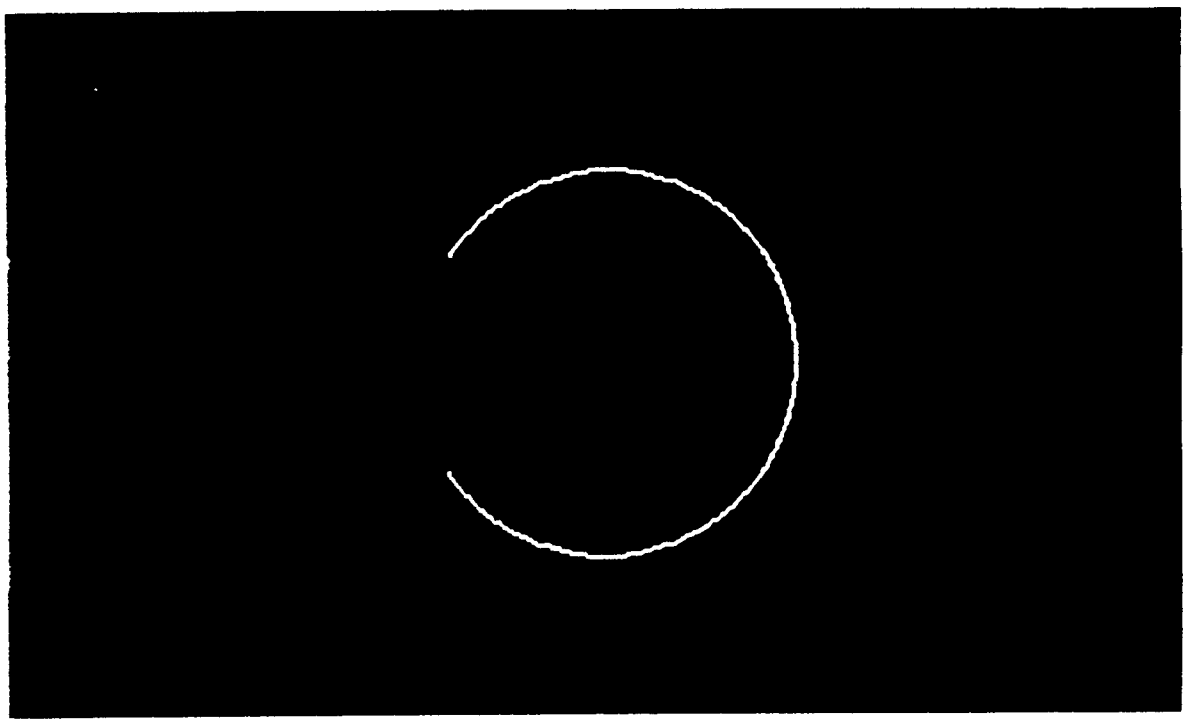


Figure 14-28 Image on the Screen Resulting from Statements 1Ø and 2Ø

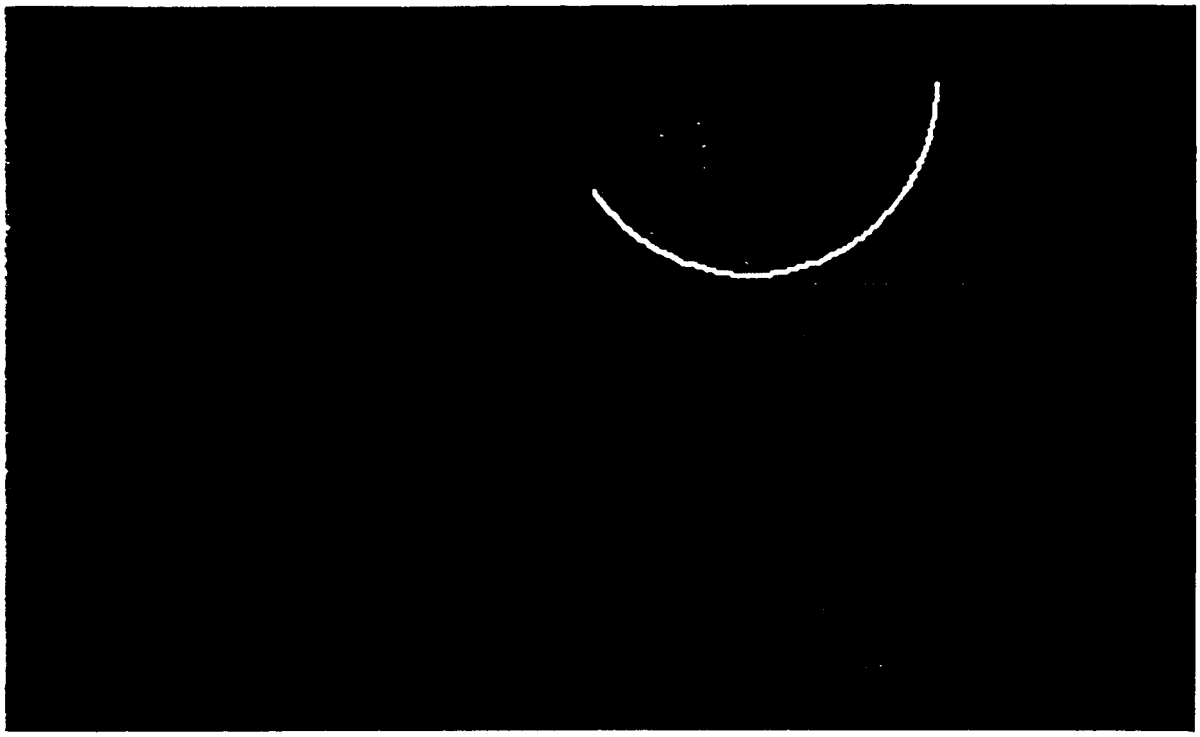


Figure 14-29 Image on the Screen Resulting from Statement 5Ø

DRAW (PROGRAM/IMMEDIATE)

Moves the virtual pen within a window and draws lines in a given colour.

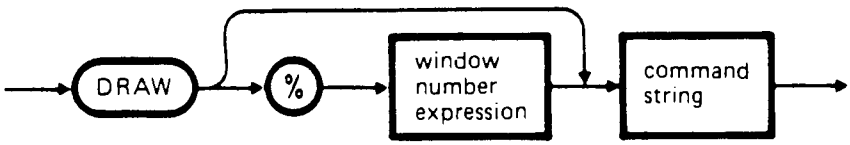


Figure 14-30 DRAW Statement

Where

SYNTAX ELEMENT	MEANING
window number expression	A numeric integer expression specifying the window in which DRAW is to operate. The default is the current window.

command string

This can be either a string constant or a string variable. The string, in both cases, consists of one or more commands shown in the table below (see Commands), which control the movement of the virtual pen.

With the exception of the C command, all commands may be prefixed with the B option, which inhibits drawing, and followed by one of the following action verbs: AND, XOR, OR, NOT, PSET, PRESET, which define an operation on any point of the line. The action verbs are specified by their first letter, except PRESET which is specified by R.

The verb P (i.e. PSET) indicates that the figure is to be drawn in the specified colour. The verbs A (i.e. AND), O (i.e. OR), and X (i.e. XOR) indicate that the colour of the figure is the result of a logical operation between the specified colour and the existing screen contents along that figure. The verb N (i.e. NOT) indicates that the colour of the figure will be the complement of the screen contents along that figure. The verb R (i.e. PRESET) indicates that the figure will be drawn in the background colour.

The default action verb is P.

Note: Command parameters (dx,dy,x,y and colour) can be expressed as variables. In this case the variable names must be written between equals signs. See the examples below.

Commands

COMMAND	MEANING
M dx,dy	Moves the pen from its present position (a,b say) to the position indicated by (a+dx,b+dy).
J x,y	Moves the pen to the position indicated by (x,y).

U dy	Moves the pen up by dy positions.
D dy	Moves the pen down by dy positions.
L dx	Moves the pen left by dx positions.
R dx	Moves the pen right by dx positions.
C colour	Sets the colour to be used to draw. A colour number must be specified after C. If no C is specified the last colour used by a preceding DRAW or the foreground colour of the current window is assumed.

Examples

DISPLAY	COMMENTS
9Ø PSET(1Ø,2Ø) 1ØØ X=23 ... 13Ø DRAW "M=X=,25" ...	Statement 9Ø colours the point (1Ø,2Ø) with the current foreground colour. Statement 1ØØ sets X=23 Statement 13Ø draws a line from the current pen position (1Ø,2Ø) to the position (33,45), that is (1Ø+23,2Ø+25)
25Ø A$="BM 1Ø,2 D 2Ø MR 15,-3"	Statement 25Ø sets A$ to the following command string: - the M command with the B option to move the pen, without drawing from its current position (a,b say) to the position (a+1Ø,b+2) - the D command to move the pen down 2Ø positions, i.e. to the point (a+1Ø,b-18) - the M command to move the pen from its current position (a+1Ø,b-18) to the point (a+25,b-21). The R option (PRESET) indicates that the line must be drawn in the current background colour.

26Ø DRAW A$	Statement 26Ø executes the sequence of commands specified by the A$ variable.

Remarks

The sequence of commands in a DRAW statement may be entered either in lower case or in upper case letters. They may be separated by blanks or they may be written contiguously.

GRAPHICS FACILITIES PROVIDED BY PCOS

The three PCOS commands LABEL, SPRINT and LSCREEN can be called in BASIC by the CALL or EXEC statements.

Using the LABEL command you can display character strings of variable sizes and orientation.

Using the SPRINT command you can print the image of either the screen or a specified window.

Using the LSCREEN command you can get a hard copy of the text contents of a specified window.

For detailed descriptions see the "Professional Computer Operating System (PCOS) User Guide".

A. ASCII CODES

This table shows decimal, hexadecimal, and binary representation of the ASCII code.

a	b	c	d	a	b	c	d	a	b	c	a	b	c
0	00	0000 0000	NUL	64	40	0100 0000	@	128	80	1000 0000	192	C0	1100 0000
1	01	0000 0001	SOH	65	41	0100 0001	A	129	81	1000 0001	193	C1	1100 0001
2	02	0000 0010	STX	66	42	0100 0010	B	130	82	1000 0010	194	C2	1100 0010
3	03	0000 0011	ETX	67	43	0100 0011	C	131	83	1000 0011	195	C3	1100 0011
4	04	0000 0100	EOT	68	44	0100 0100	D	132	84	1000 0100	196	C4	1100 0100
5	05	0000 0101	ENQ	69	45	0100 0101	E	133	85	1000 0101	197	C5	1100 0101
6	06	0000 0110	ACK	70	46	0100 0110	F	134	86	1000 0110	198	C6	1100 0110
7	07	0000 0111	BEL	71	47	0100 0111	G	135	87	1000 0111	199	C7	1100 0111
8	08	0000 1000	BS	72	48	0100 1000	H	136	88	1000 1000	200	C8	1100 1000
9	09	0000 1001	HT	73	49	0100 1001	I	137	89	1000 1001	201	C9	1100 1001
10	0A	0000 1010	LF	74	4A	0100 1010	J	138	8A	1000 1010	202	CA	1100 1010
11	0B	0000 1011	VT	75	4B	0100 1011	K	139	8B	1000 1011	203	CB	1100 1011
12	0C	0000 1100	FF	76	4C	0100 1100	L	140	8C	1000 1100	204	CC	1100 1100
13	0D	0000 1101	CR	77	4D	0100 1101	M	141	8D	1000 1101	205	CD	1100 1101
14	0E	0000 1110	SO	78	4E	0100 1110	N	142	8E	1000 1110	206	CE	1100 1110
15	0F	0000 1111	SI	79	4F	0100 1111	O	143	8F	1000 1111	207	CF	1100 1111
16	10	0001 0000	DLE	80	50	0101 0000	P	144	90	1001 0000	208	D0	1101 0000
17	11	0001 0001	DC1	81	51	0101 0001	Q	145	91	1001 0001	209	D1	1101 0001
18	12	0001 0010	DC2	82	52	0101 0010	R	146	92	1001 0010	210	D2	1101 0010
19	13	0001 0011	DC3	83	53	0101 0011	S	147	93	1001 0011	211	D3	1101 0011
20	14	0001 0100	DC4	84	54	0101 0100	T	148	94	1001 0100	212	D4	1101 0100
21	15	0001 0101	NAK	85	55	0101 0101	U	149	95	1001 0101	213	D5	1101 0101
22	16	0001 0110	SYN	86	56	0101 0110	V	150	96	1001 0110	214	D6	1101 0110
23	17	0001 0111	ETB	87	57	0101 0111	W	151	97	1001 0111	215	D7	1101 0111
24	18	0001 1000	CAN	88	58	0101 1000	X	152	98	1001 1000	216	D8	1101 1000
25	19	0001 1001	EM	89	59	0101 1001	Y	153	99	1001 1001	217	D9	1101 1001
26	1A	0001 1010	SUB	90	5A	0101 1010	Z	154	9A	1001 1010	218	DA	1101 1010
27	1B	0001 1011	ESC	91	5B	0101 1011	[	155	9B	1001 1011	219	DB	1101 1011
28	1C	0001 1100	FS	92	5C	0101 1100	\	156	9C	1001 1100	220	DC	1101 1100
29	1D	0001 1101	GS	93	5D	0101 1101	]	157	9D	1001 1101	221	DD	1101 1101
30	1E	0001 1110	RS	94	5E	0101 1110	^	158	9E	1001 1110	222	DE	1101 1110
31	1F	0001 1111	US	95	5F	0101 1111	_	159	9F	1001 1111	223	DF	1101 1111
32	20	0010 0000	SPACE	96	60	0110 0000	`	160	A0	1010 0000	224	E0	1110 0000
33	21	0010 0001	!	97	61	0110 0001	a	161	A1	1010 0001	225	E1	1110 0001
34	22	0010 0010	"	98	62	0110 0010	b	162	A2	1010 0010	226	E2	1110 0010
35	23	0010 0011	#	99	63	0110 0011	c	163	A3	1010 0011	227	E3	1110 0011
36	24	0010 0100	$	100	64	0110 0100	d	164	A4	1010 0100	228	E4	1110 0100
37	25	0010 0101	%	101	65	0110 0101	e	165	A5	1010 0101	229	E5	1110 0101
38	26	0010 0110	&	102	66	0110 0110	f	166	A6	1010 0110	230	E6	1110 0110
39	27	0010 0111	'	103	67	0110 0111	g	167	A7	1010 0111	231	E7	1110 0111
40	28	0010 1000	(	104	68	0110 1000	h	168	A8	1010 1000	232	E8	1110 1000
41	29	0010 1001	)	105	69	0110 1001	i	169	A9	1010 1001	233	E9	1110 1001
42	2A	0010 1010	*	106	6A	0110 1010	j	170	AA	1010 1010	234	EA	1110 1010
43	2B	0010 1011	+	107	6B	0110 1011	k	171	AB	1010 1011	235	EB	1110 1011
44	2C	0010 1100	,	108	6C	0110 1100	l	172	AC	1010 1100	236	EC	1110 1100
45	2D	0010 1101	-	109	6D	0110 1101	m	173	AD	1010 1101	237	ED	1110 1101
46	2E	0010 1110	.	110	6E	0110 1110	n	174	AE	1010 1110	238	EE	1110 1110
47	2F	0010 1111	/	111	6F	0110 1111	o	175	AF	1010 1111	239	EF	1110 1111
48	30	0011 0000	0	112	70	0111 0000	p	176	B0	1011 0000	240	F0	1111 0000
49	31	0011 0001	1	113	71	0111 0001	q	177	B1	1011 0001	241	F1	1111 0001
50	32	0011 0010	2	114	72	0111 0010	r	178	B2	1011 0010	242	F2	1111 0010
51	33	0011 0011	3	115	73	0111 0011	s	179	B3	1011 0011	243	F3	1111 0011
52	34	0011 0100	4	116	74	0111 0100	t	180	B4	1011 0100	244	F4	1111 0100
53	35	0011 0101	5	117	75	0111 0101	u	181	B5	1011 0101	245	F5	1111 0101
54	36	0011 0110	6	118	76	0111 0110	v	182	B6	1011 0110	246	F6	1111 0110
55	37	0011 0111	7	119	77	0111 0111	w	183	B7	1011 0111	247	F7	1111 0111
56	38	0011 1000	8	120	78	0111 1000	x	184	B8	1011 1000	248	F8	1111 1000
57	39	0011 1001	9	121	79	0111 1001	y	185	B9	1011 1001	249	F9	1111 1001
58	3A	0011 1010	:	122	7A	0111 1010	z	186	BA	1011 1010	250	FA	1111 1010
59	3B	0011 1011	;	123	7B	0111 1011	{	187	BB	1011 1011	251	FB	1111 1011
60	3C	0011 1100	<	124	7C	0111 1100	|	188	BC	1011 1100	252	FC	1111 1100
61	3D	0011 1101	=	125	7D	0111 1101	}	189	BD	1011 1101	253	FD	1111 1101
62	3E	0011 1110	>	126	7E	0111 1110	~	190	BE	1011 1110	254	FE	1111 1110
63	3F	0011 1111	?	127	7F	0111 1111	DEL	191	BF	1011 1111	255	FF	1111 1111

a Decimal

b Hexadecimal

c 8 bit Binary Representation

d ASCII Code

Note: Boxed characters are different on national keyboards (see Appendix B).

B. ASCII CHARACTER EQUIVALENCES

This table shows the national equivalences for those ASCII characters which appear on the video screen or printer in various national guises.

ASCII VALUE		NATIONAL EQUIVALENT														
DECIMAL	HEXADECIMAL	USA	ITALY	FRANCE	GREAT BRITAIN	GERMANY (1)	GERMANY (2)	SPAIN	PORTUGAL	DENMARK	SWEDEN FINLAND	NORWAY	SWITZERLAND FRENCH	SWITZERLAND GERMAN	GREECE	YUGOSLAVIA
35	23	#	£	£	£	#	#	£	#	£	#	£	£	£	£	#
36	24	$	$	$	$	$	$	$	$	$	¤	$	$	$	$	¤
64	40	@	§	à	@	§	§	§	§	´	É	´	§	§	@	§
91	5B	[	°	°	[	Ä	Ä	¡	Ã	Æ	Ä	Æ	à	à	[	Ð
92	5C	\	ç	ç	\	Ö	Ö	Ñ	Ç	Ø	Ö	Ø	ç	ç	\	¨
93	5D	]	é	§	]	Ü	Ü	¿	Õ	Å	Å	Å	è	è	]	Ž
96	60	`	ù	`	`	`	`	`	`	`	`	`	`	`	`	š
123	7B	{	à	é	{	a	a	ª	ã	æ	a	æ	a	a	{	ð
124	7C	|	ò	ù	|	o	o	Ñ	ç	ø	o	ø	o	o	|	ć
125	7D	}	è	è	}	u	u	ç	õ	a	a	a	u	u	}	ž
126	7E	~	ì		~	ß	ß	~	ª		—		é	é		č

* Encircled characters are used for functions in BASIC.

C. ERROR CODES AND THEIR MEANING

ABOUT THIS APPENDIX

This Appendix lists all the errors returned from the BASIC Interpreter. They are not displayed with their error number: only the description is displayed.

CONTENTS

ERROR CODES AND THEIR MEANING

ERROR CODE	MESSAGE	COMMENT
1	NEXT without FOR	A NEXT statement has been encountered without a matching FOR
2	Syntax error	A line has been encountered which includes an incorrect sequence of characters (misspelled keyword, incorrect punctuation etc.)
3	RETURN without GOSUB	A RETURN has been encountered for which there is no previous unmatched GOSUB statement
4	Out of DATA	A READ statement has been executed when there are no DATA statements with unread data remaining in the program
5	Illegal function call	A parameter that is out of range has been passed to a numeric or a string function. Such an error may occur when: a. An array subscript is either negative or too big. b. A log function is assigned a negative or a null argument. c. The SQR function is assigned a negative value. d. A negative number has an exponent which is not an integer. e. A USR function has been called without having established the initial address. f. An incorrect argument has been made in one of the following functions: MID$, LEFT$, RIGHT$, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.

ERROR CODE	MESSAGE	COMMENT
6	Overflow	The result of a calculation is too large to be represented in BASIC's number format. N.B. With underflow, the result is taken as zero, and execution continues without indication of an error.
7	Out of memory (also used in PCOS)	A program is too big; or has too many loops, GOSUBS, variables; or has expressions too complicated to evaluate
8	Undefined line number	A line reference is to a non-existent line from a GOTO, GOSUB, IF..THEN..ELSE or DELETE
9	Subscript out of range	An array element has been referred to either with a subscript that is outside the dimensions of the array or with the wrong number of subscripts
1Ø	Duplicate Definition	Two DIM statements have been given for the same array, or a DIM statement has also been applied to an array after the default dimension of 10 was previously established for that array
11	Division by zero	A division by zero has been encountered or the value zero has been raised to a negative power. In the former case the result is machine infinity (with the appropriate sign) and in the latter case the result is positive machine infinity

ERROR CODES AND THEIR MEANING

ERROR CODE	MESSAGE	COMMENT
12	Illegal direct	A statement which is invalid in immediate (direct) mode has been entered as an immediate command.
13	Type mismatch (also used in PCOS)	A string variable name has been assigned a numeric value or vice versa; a function that expects a numeric argument has been given a string argument or vice versa
14	Out of string space	String variables have caused BASIC to exceed the amount of free user memory remaining. (BASIC will allocate space dynamically until it runs out of memory)
15	String too long	An attempt has been made to create a string more than 255 characters long
16	String formula too complex	A string expression is too long or too complex to be processed. It should be broken into smaller expressions
17	Can't continue	An attempt has been made to continue a program that is non-continuable; as it is halted due to an error, was modified during a break in execution, or does not exist in user memory
18	Undefined user function	A function, that has not been previously defined, has been called
19	No RESUME	An error-trapping routine has been entered that contains no RESUME statement
2Ø	RESUME without error	A RESUME statement has been encountered before an error-trapping routine is entered

ERROR CODE	MESSAGE	COMMENT
22	Missing operand	An expression contains an operator but no following operand
23	Line buffer overflow	An attempt has been made to enter a line with more than 255 characters
26	FOR without NEXT	A FOR has been encountered without a matching NEXT
29	WHILE without WEND	A WHILE has been encountered without a matching WEND
3Ø	WEND without WHILE	A WEND has been encountered without a matching WHILE
31	IEEE: Invalid talker/ listener address	Use of illegal talker listener address
32	IEEE: talker = listener address	An attempt has been made to talk to a talker, or listen to a listener
33	IEEE: Unprintable error	An error message is not printable i.e. corresponds to an error with an undefined error code
34	IEEE: Board not present	An attempt has been made to use IEEE on a machine which does not have the optional IEEE interface
35	Window not open	An attempt has been made to use a window which is not at present open
36	Unable to create window	The window to be created is too big or too small for its mode (graphics or text)
37	Invalid action verb	An action verb has been incorrectly spelt or used
38	Parameter out of range	One or more parameters have exceeded the limits set for their range

ERROR CODES AND THEIR MEANING

ERROR CODE	MESSAGE	COMMENT
39	Too many dimensions	An attempt has been made to use an array of more than one dimension, in graphics mode
4Ø	PCOS error	A PCOS error has been detected when a PCOS command is being called from BASIC
5Ø	FIELD overflow	A FIELD statement has attempted to allocate more bytes than were specified for the record length of a random file
51	Internal error (also used in PCOS)	An internal malfunction has occured. Report the conditions under which the error occurred to your Support Organisation
52	Bad file number	A statement or command refers to a file (having a file number not within the range specified at initialization) or the corresponding file is not open
53	File not found (also used in PCOS)	A LOAD, KILL or OPEN statement refers to a file that does not exist on the current disk
54	Bad file mode (also used in PCOS)	An attempt has been made to use random file operations (GET# or PUT#) with a sequential file; or to use the sequential operation LOAD with a random file; or to use an illegal file mode with OPEN, i.e. not A,I,O, or R
55	File already open (also used in PCOS)	A sequential OPEN, O has been issued for a file that is already open, or a KILL has been applied to a file that is open

ERROR CODE	MESSAGE	COMMENT
57	Disk I/O error (also used in PCOS)	An input/output error has occurred during a disk I/O operation. It is a termination error, i.e. PCOS/-BASIC cannot recover - apply a RESET
58	File already exists (also used in PCOS)	The filename specified in a NAME statement is identical to a file-name already in use on the disk
61	Disk full (also used in PCOS)	All disk storage space available is in use
62	Input past end	An INPUT statement has been exe-cuted: after all the data has been assigned, or for an empty (null) file. Hint: use EOF function to detect end of file
63	Bad record number	The record number used with a GET or PUT statement exceeds range, i.e. is Ø or greater than 32767
64	Bad file name (also used in PCOS)	An invalid form of filename has been used with KILL, LOAD, OPEN or SAVE e.g.: - too long - includes illegal characters such as space or hyphen
66	Direct statement in file	A direct (immediate) statement has been encountered when loading an ASCII format file. The LOAD operation is terminated
67	Too many files (also used in PCOS)	An attempt has been made to create a new file (using SAVE or OPEN) when the present directory is already full

ERROR CODES AND THEIR MEANING

ERROR CODE	MESSAGE	COMMENT
69	Volume name not found	The volume name referred to does not match (either of) the disk(s) currently inserted
7Ø	Rename error	An attempt has been made to rename a volume with an illegal name
71	Volume number error	The specified volume number is illegal
72	Volume not enabled	The volume identifier includes a password which must be quoted
73	Invalid password	The password entered is illegal
74	Illegal disk change	The disk has been changed since last using the file
75	Write Protected	An attempt has been made to write to a write protected disk
76	Error in Parameter	A parameter contains an illegal character
77	Too many parameters	More than the required number of parameters have been specified
78	File not OPEN	An attempt has been made to access a file that is not open

D. DIFFERENCES BETWEEN PCOS RELEASES AFFECTING BASIC

ABOUT THIS APPENDIX

This Appendix shows the differences between 1.3 and 2.Ø PCOS releases affecting BASIC.

CONTENTS

DIFFERENCES BETWEEN PCOS RELEASES AFFECTING BASIC

PCOS RELEASE 1.3	PCOS RELEASE 2.Ø
Hard Disk is not supported	Hard Disk is supported. With a hard disk system drive numbers are: Ø (on the left, to insert diskettes) 1Ø (on the right, where the hard disk is mounted)
The 16ØK byte diskettes are not supported	The 16ØK byte diskettes are supported
8-colour video is not supported	8-colour video is supported. With an 8-colour display the colour numbers coincide with the colour codes and the COLOR (global colour set selection) statement, if used, does not produce any effect
The Assembly language is not supported	The Assembly language is supported
The following PCOS commands are not supported:	The following PCOS commands are supported:
ASM, BVOLUME, CKEY, DCONFIG, LSCREEN, PDEBUG, PUNLOAD, RFONT, SLANG, TLOC, VVERIFY, WFONT	ASM, BVOLUME, CKEY, DCONFIG, LSCREEN, PDEBUG, PUNLOAD, RFONT, SLANG, TLOC, VVERIFY, WFONT
Greece and Yugoslavia keyboards are not supported	Greece and Yugoslavia keyboards are supported
BASIC command is resident	BASIC command is transient
PCOS and BASIC are booted at initialization	Only PCOS is booted at initialization

PCOS RELEASE 1.3	PCOS RELEASE 2.Ø
At initialization the last selected drive is drive Ø	At initialization the last selected drive is the one that PCOS is booted from
The PCOS prompt is >	The PCOS prompt is n> where n specifies the last selected drive, and may be: Ø, 1 } (with 2-diskette systems) Ø, 1Ø } (with hard-disk systems)
The default value of the memory parameter in the CLEAR statement is 38ØØØ	The default value of the memory parameter in the CLEAR statement is 37ØØØ
The LABEL PCOS command does not permit a colour parameter, thus label strings may only be drawn with the foreground colour	The LABEL PCOS command may specify a colour parameter, thus label string may be drawn in a specified colour
The range of values of the "position" parameter in the WINDOW (To open a window) statement, if a horizontal split is to be made, is: lower limit = vertical spacing value of the parent window + 1 upper limit = height of the parent window - (lower limit + 1)	The range of values of the "position" parameter in the WINDOW (To open a window) statement, if a horizontal split is to be made, is: lower limit = 1 upper limit = 255

E. BASIC STATEMENTS, COMMANDS AND FUNCTIONS

ABOUT THIS APPENDIX

This Appendix lists all BASIC statements, commands and functions in alphabetical order and provides a reference to the corresponding page.

If a statement or a command may be used both in a program and an immediate line, PROGRAM/IMMEDIATE is specified. If a command may only be used in an immediate line, IMMEDIATE is specified. If a statement may only be used in a program line, PROGRAM is specified.

Instead for a function nothing is mentioned, as functions may always be written both in a program and an immediate line.

CONTENTS

BASIC STATEMENTS, COMMANDS AND FUNCTIONS

BASIC STATEMENTS, COMMANDS AND FUNCTIONS

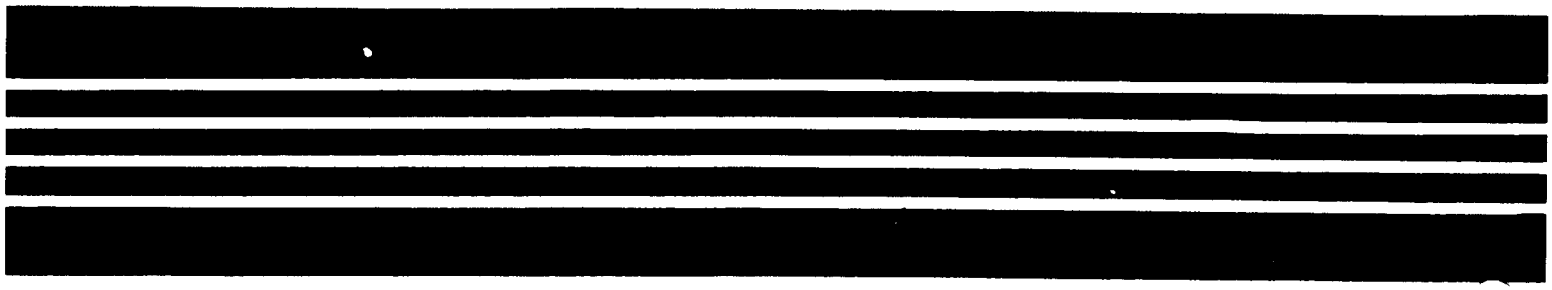
BASIC STATEMENTS, COMMANDS AND FUNCTIONS

NOTICE

Ing. C. Olivetti & C. S.p.A. reserves the right to make improvements in the product described in this manual at any time and without notice.

This material was prepared for the benefit of Olivetti customers. It is recommended that the package be test run before actual use.

Anything in the standard form of the Olivetti Sales Contract to the contrary not withstanding, all software being licensed to Customer is licensed "as is". THERE ARE NO WARRANTIES EXPRESS OR IMPLIED INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTY OF FITNESS FOR PURPOSE AND OLIVETTI SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL OR INCIDENTAL DAMAGES IN CONNECTION WITH SUCH SOFTWARE.



GR Code 3982430 P (2)
Printed in Italy