BASIC KEYWORDS FOR THE APPLE® ///

DIE ADAMIS

BASIC Keywords for the Apple III

BASIC Keywords for the Apple III

Eddie Adamis

A Wiley Press Book

John Wiley & Sons, Inc.

New York • Chichester • Brisbane • Toronto • Singapore

Library of Congress Cataloging in Publication Data

Adamis, Eddie, 1921–
Basic keywords for the APPLE III.

Includes index.
1. Apple III (Computer)—Programming. 2. Basic (Computer program language) I. Title.
QA76.8.A663A3 1983 001.64′2 83-12327
ISBN 0-471-88389-1

Printed in the United States of America

84 85 10 9 8 7 6 5 4 3 2 1

Contents

Foreword

Eddie Adamis has, in my view, fulfilled the dearest wishes of the founders of Apple: To open the world of personal computing to the nonspecialist. He has, in fact, a talent that is all too rare: the ability to take something considered obscure—and even a little frightening—and make it clear and simple.

The author's history highlights the source of this talent. He came to personal computing by the most improbable route: composer, music arranger, Managing Director of United Artists Music and Records (France) for fourteen years. His passion for personal computing started when he was fifty. His acquisition of skill and enthusiasm has not dulled his memory; he writes now as he wishes others had written for him when he was just learning.

BASIC Keywords for the Apple III explores progressively and thoroughly the Business BASIC language of the Apple III. Each instruction is described, with its variations, through clear and precise examples.

Eddie Adamis brings two important extras to technical manuals:

- his own viewpoint—not having been involved with the development of the language, Eddie Adamis approaches Business BASIC with a fresh eye;
- not being a computer man by trade, he writes for other nonspecialists who want to use the personal computer for their own businesses, with a sympathy that is obvious from his attention to detail in making everything simple.

By now, everyone will have gathered that I highly recommend this book.

Eddie Adamis will make your Apple III and Business BASIC even better.

Jean-Louis Gassée
President, Apple Computer France

Preface

Since its creation in the 1960s at Dartmouth College by John G. Kemeny and Thomas E. Kurtz, the popularity of the BASIC language has never stopped growing. This is, first, because BASIC is easy to learn and understand and, second, because its flexibility and power are such that it has given birth to numerous "extensions" specifically designed for particular systems.

This book is organized in the form of a dictionary, which allows the reader to refer quickly to the instructions, commands, operators, and symbols of Business BASIC for the Apple III. The keywords, all the symbols and operators, are presented one to a page. Each presentation provides:

- the meaning of the keyword
- its working principle
- a guideline for its use
- a program example
- the results of the executed program

and practical comments on the keyword, its use, difficulties, and the like.

The book is written in the clearest and most concise way possible, with a consistent visual presentation, to provide an introduction to and a tutorial in BASIC programming in general. Reading it does not require any specialized knowledge. I have deliberately avoided filling the text with heavy technical explanations specific to the system, with the idea that the interested reader will be able to refer to the relevant manuals and/or user's guides to the Apple III.

Syntax Notation

Business BASIC keywords are written in uppercase letters.

Example: CLEAR

Keywords,

Example: CHAIN *pathname* [, *line number*]

delimiters (punctuation marks),

Example: CHAIN *pathname* [**,** *line number*]

and special characters appended to keywords and/or variable names

Example: LEFT**$**
LIST**&**
AREA**%**

must be typed exactly as shown.

Information that you must fill in is represented in lowercase letters in *italics*.

Example: CHAIN *pathname* [, *line number*]

Format descriptions may consist of one or more compound elements.

Symbols used to describe compound elements syntax are:

| to separate alternative elements;

Example: CREATE *pathname*, CATALOG|TEXT|DATA

[ ] to enclose optional elements;

Example: CHAIN *pathname* [, *line number*]

{ } to enclose repeatable elements that must occur at least once.

Example: INPUT] *variable* {, *variable*}

The above symbols must not be typed in. They are used only to set off the elements that are alternative, optional, and repeatable.

BASIC Keywords for the Apple III

ABS

stands for **ABSOLUTE**

TYPE Numeric function

FORMAT **ABS** (*arithmetic expression*)

ACTION Returns the absolute value of a numeric expression.

The absolute value of a number is always positive or zero. A negative value is converted to the equivalent positive value.

Numeric functions may be used either in immediate mode in conjunction with a PRINT statement or in deferred execution. The argument to all numeric functions must be an *arithmetic expression.* All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions.

EXAMPLE

1. *arithmetic expression* can be a numeric constant;

PRINT **ABS** (0)	Returns 0
PRINT **ABS** (10)	Returns 10
PRINT **ABS** (−10)	Returns 10

2. a numeric variable;

A = −25.65 : PRINT **ABS** (A)	Returns 25.65
B = 30.36 − 40 : PRINT **ABS** (B)	Returns 9.64

3. an arithmetic operation;

PRINT 10 + **ABS** (−20.36)	Returns 30.26
PRINT **ABS** (−12 * 6)	Returns 72

4. any valid combination thereof.

A = −25.65 : B = 30.36−40	
PRINT 10 + **ABS** (A + B + (−12 * 6))	Returns 117.29

NOTES

- Business BASIC has 16 numeric functions in the following type categories:

trigonometric:	ATN, COS, SIN, TAN
arithmetic:	**ABS**, EXP, INT, LOG, RND, SGN, SQR
conversion:	CONV, CONV%, CONV&, CONV$
user-defined:	DEF FN

ADDITION

symbol +

TYPE Arithmetic operator

FORMAT *numeric expression1 + numeric expression2*

ACTION Performs arithmetic addition.

EXAMPLE

1. *numeric expression* can be a numeric constant;

PRINT 20 + 15	Returns 35
PRINT 20 + 10 + 5	Returns 35
PRINT 20 + (−25)	Returns −5

2. a numeric variable;

A = 20 : B = 15 : C = 10 : D = 5 : E = −25

PRINT A + B	Returns 35
PRINT A + C + D	Returns 35
PRINT A + E	Returns −5

3. any valid combination thereof.

A = 20 : B = 15 : C = 10 : D = 5 : E = −25

PRINT A + 10 + D	Returns 35
PRINT 20 + C + D	Returns 35
PRINT A + E	Returns −5

NOTES

- Business BASIC has 9 arithmetic operators:

+	Unary plus
−	Unary minus
^	Exponentiation
*	Multiplication
/	Floating-point division
MOD	Modulo division
DIV	Integer division
+	Addition
−	Subtraction

AMPERSAND

symbol **&**

TYPE Identifier

FORMAT *variable name* **&**

ACTION Identifies the variable as being of the long integer type.

Variables have identifiers attached to specify which type of value they represent. A variable without an identifier is automatically of the single-precision type.

EXAMPLE

```
Sales&
TOTAL.SALES.1983&
Number.of.Items&
```

NOTES

- Variable names must always begin with a letter. You can have from 0 (zero) to 63 additional characters after the first letter. The additional characters can only be letters, digits, or periods. Long integer variables may not be mixed in arithmetic expressions with regular integers or reals. In variable names, lowercase letters are considered equivalent to their uppercase counterparts.
- A *long integer* is any positive or negative whole number without a decimal point. It has eight or more digits (up to 19). Its value is within the range from −9223372036854775808 to 9223372036854775807. A value greater than 9223372036854775807 would cause the ?OVERFLOW ERROR message to be displayed.
- Business BASIC has three identifiers attached to variable names:

&	For variables of the long integer type
%	For variables of the integer type
$	For variables of the string type

AND

TYPE Logical operator

FORMAT *condition1* **AND** *condition2*

ACTION Connects two or more conditions.

The expression evaluates as true (non-zero) if both conditions are true; otherwise, it evaluates as false (zero). The result of the evaluation is then usually used in conditional statements, such as IF ... THEN statements, to make a decision regarding program flow.

EXAMPLE

```
10  A = 10 : B = 50 : C = 100
20  IF A < B AND C > B THEN 40
30  PRINT "THE RESULT OF THE EVALUATION IS FALSE" :
    END
40  PRINT "BOTH OF THE CONDITIONS HAVE BEEN MET"
50  A$ = "A" : B$ = "B" : C$ = "C"
60  IF A$ < > B$ AND C$ < > B$ THEN 80
70  PRINT "THE RESULT OF THE EVALUATION IS FALSE" :
    END
80  PRINT "BOTH OF THE CONDITIONS HAVE BEEN MET"
90  END
```

RESULT

Line 40: Both of the conditions have been met: A is less than B and C is greater than B; the message on line 40 is printed.

80: Both of the conditions have been met: A is different from B and C is different from B; the message on line 80 is printed.

NOTES

- The strings are compared character by character, from left to right, on the basis of their ASCII code numbers. The first character found in one string that has a greater ASCII value than the character found in the same position in the second string makes the first string greater. If the characters in the same positions are identical but one string's current length is longer, the longer string is greater.
- Business BASIC has three logical operators:

AND	Conjunction
OR	Inclusive disjunction
NOT	Negation (logical complement)

ASC

stands for **ASCII**

TYPE String function

FORMAT **ASC** (*string expression*)

ACTION Returns the ASCII numeric code for the first character of a string expression.

EXAMPLE

1. *string expression* can be a string constant (literal);

   PRINT **ASC** ("A") Returns 65
   PRINT **ASC** ("ADAM") Returns 65

2. a string variable;

   A$ = "A" : B$ = "ADAM"
   PRINT **ASC** (A$) Returns 65
   PRINT **ASC** (B$) Returns 65

3. a substring function;

   A$ = "ADAM"
   PRINT **ASC** (LEFT$(A$,1)) Returns 65

4. any valid combination thereof.

   A$ = " AFTERNOON"
   PRINT **ASC** (MID$("GOOD" + A$,6,1)) Returns 65

NOTES

- 65 is the ASCII numeric code for a capital A.
- ASCII stands for American Standard Code for Information Interchange.
- The number of characters in a string expression may range from 0 (zero) to 255.
- A null string is a string that contains no characters.
- A string variable is identified by a dollar sign ($).
- The CHR$ function is the inverse of the ASC function. It converts the ASCII code to a character.
- Business BASIC has 12 string or string-related functions: **ASC**, CHR$, HEX$, INSTR, LEFT$, LEN, MID$, RIGHT$, STR$, SUB$, TEN, VAL.

AS EXTENSION

TYPE File clause

FORMAT OPEN# *file number* **AS EXTENSION**, *file name*

ACTION Appends information at the end of a file.

With an **AS EXTENSION** clause, PRINT# or WRITE# statements write additional information beginning at the end of the open file, thus allowing the user to retain information previously saved in the file. The first access begins at the end of the existing file. Each subsequent access begins where the last one left off.

EXAMPLE

```
10  OPEN#1 AS EXTENSION, Accounting
20  FOR X = 60000 TO 60100
30  PRINT#1; "Account number ";X
40  PRINT#1; " Pending "
50  NEXT X
60  CLOSE#1
70  END
```

RESULT

Line 10: Opens file #1 with the **AS EXTENSION** clause.

20: Sets up a loop to repeat 100 times.

30: Prints the heading "Account number" followed by the value of X.

40: Prints the message.

50: Repeats from line 20.

60: Closes file #1.

NOTES

- The comma that is usually placed after the file reference number in a regular OPEN# statement is moved to the right of the clause.
- Business BASIC has three file clauses: AS INPUT, AS OUTPUT, **AS EXTENSION**.

AS INPUT

TYPE File clause

FORMAT OPEN# *file number* **AS INPUT**, *file name*

ACTION Specifies that the opened file is a read-only file.

EXAMPLE

```
 10  REM *** Displaying with an INPUT# statement
 20  REM *** the contents of a sequential text file
 30  ST$ = "Sequential Text"
 40  OPEN#1 AS INPUT,ST$
 50  ON EOF#1 GOTO 100
 60  INPUT#1; L$
 70  PRINT#1 L$
 80  GOTO 40
 90  CLOSE#1
100  END
```

RESULT

Line 10–20: Remarks to document program.

30: Assigns a file name to the string variable ST$.

40: Opens the named file as a read-only file and assigns to it #1 as its reference number.

50: Branches unconditionally to line 100 when the end-of-file marker is reached. (EOF is a reserved variable that stands for end of file.)

60: Reads a line of text and assigns it to the string variable L$.

70: Displays the line on the screen. The numeric value of X, which was previously converted to a string, will, this time, be converted back to a numeric value and displayed with a space in front of it, as usual for any positive numeric expression.

80: Branches back to line 40. INPUT# and PRINT# will keep on reading and writing, respectively, until the end of the file is reached.

90: Closes file #1.

NOTES

- You cannot write to a file after the **AS INPUT** option has been executed.
- Business BASIC has three file clauses: **AS INPUT**, AS OUTPUT, AS EXTENSION.

AS OUTPUT

TYPE File clause

FORMAT OPEN# *file number* **AS OUTPUT**, *file name*

ACTION Specifies that the opened file is a write-only file.

EXAMPLE

```
10 REM *** Writing with a PRINT# statement both string and numeric
20 REM *** values into a sequential text file
30 ST$ = "Sequential Text"
40 OPEN#1 AS OUTPUT,ST$
50 FOR X = 1 TO 10
60 PRINT#1; "Text line number ";X
70 NEXT X
80 CLOSE#1
90 END
```

RESULT

Line 10–20: Remarks to document program.

30: Assigns a file name to the string variable ST$.

40: Opens the named file as a write-only file and assigns to it #1 as its reference number.

50: Sets up a loop to execute 10 times.

60: Writes to the file the string "Text line number" followed by the numeric value of X (1 through 10) automatically converted to a string. These two strings, concatenated because of the semicolon, will occupy one line of text in the file.

70: Branches back to line 50 (loop to execute 10 times).

80: Closes file #1.

NOTES

- You cannot read from a file after the **AS OUTPUT** option has been executed.
- Business BASIC has three file clauses: AS INPUT, **AS OUTPUT**, AS EXTENSION.

ASSIGNMENT *symbol =*

TYPE Operator

FORMAT *variable* | *reserved variable* = *value*

ACTION Assigns *value* to the variable specified by *variable name*.

EXAMPLE

```
10  A = 10
20  B = A + 10
30  C = (A * B)/2
40  L$ = "THE BASIC LANGUAGE"
50  PRINT A,B,C,L$
60  END
```

RESULT

Line 10: Variable A is assigned the value 10.

20: Variable B is assigned the result of the addition.

30: Variable C is assigned the result of the mathematical operation.

40: Variable L$ is assigned the string THE BASIC LANGUAGE.

50: The four variables' values are printed out.

NOTES

- The keyword LET is optional.

  Example: LET *variable name* = *value*

  and

  variable name = *value*

  are equivalent statements.

  Although *variable name* = *value* looks like a relational expression, it is interpreted by Business BASIC as an assignment statement, and has no logical value.

ATN

stands for **ARC TANGENT**

TYPE Numeric function

FORMAT **ATN** (*arithmetic expression*)

ACTION Returns the arc tangent of *arithmetic expression.*

Numeric functions may be used either in immediate mode in conjunction with a PRINT statement or in deferred execution. The argument to all numeric functions must be an *arithmetic expresssion.* All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions.

EXAMPLE

```
10  REM *** OS = Side opposite to angle A
20  REM *** AS = Side adjacent to angle A
30  REM *** A = Angle of a right triangle
40  OS = 6 : AS = 8
50  R = OS/AS : PRINT R
60  A = ATN (R) : PRINT A
70  END
```

RESULT

Line 10–30: Remarks to document program.

40: Assigns values to variables.

50: Prints the result: .75.

60: Prints the result: .643501109.

NOTES

- Tangent is the opposite of arc tangent. TAN (A) = OS/AS. The **ATN** function returns the angle whose tangent is *arithmetic expression.* The result is a value expressed in radians.
- *Conversions:*
  Radian = Degree / 57.29577951
  Degree = Radian * 57.29577951
- Business BASIC has 16 numeric functions in the following type categories:

trigonometric:	**ATN**, COS, SIN, TAN
arithmetic:	ABS, EXP, INT, LOG, RND, SGN, SQR,
conversion:	CONV, CONV%, CONV&, CONV$
user-defined:	DEF FN

CATALOG

TYPE File statement

FORMAT **CAT[ALOG]**

ACTION Displays a listing (names of all files) of a root directory or subdirectory specified by either a volume name or a subdirectory.

A listing of a root directory or subdirectory displayed by **CAT[ALOG]** specifies for each listed file: the size (number of blocks); the date and time of modification, the EOF standing for end of file, and the type of the file.

EXAMPLE

CATALOG
CATALOG/Memories
CATALOG/Memories/Part.One
CATALOG/D1

NOTES

- **CATALOG** may optionally be abbreviated as **CAT**.
- The file types are:

BASIC	BASIC program created with the SAVE command
BINARY	Assembly language
CAT	Root directory or subdirectory
DATA	BASIC data
FONT	Binary information about a character set
FOTO	Data representing a picture
PASCOD	Pascal code
PASDTA	Pascal data
PASTXT	Pascal text
RESERV	Reserved for future types
TEXT	BASIC text
UNKNWN	Stands for unknown; BASIC data or text file opened but not written to

CHAIN

TYPE File statement

FORMAT **CHAIN** *pathname* [, *line number*]

ACTION Loads and runs one or more specified programs.

When a program is too large (that is, when it requires more memory than is available), it may be split into sections and saved on disk. Then automatic execution of each section of the original program is performed with the **CHAIN** statement.

EXAMPLE

```
10  REM *** Accounting.Section.One
20  X = 100
30  PRINT X
40  CHAIN ".D2/Accounting.Section.Two"
```

```
10  REM *** Accounting.Section.Two
20  X = X + 100
30  PRINT X : END
40  X = X + 1000
50  PRINT X
60  END
```

RESULT

After LOADing into the conputer's memory and RUNning Accounting.Section.One, program execution proceeds as follows:

Line 30: PRINT displays the assigned value to variable X at line 30, that is, 100.

40: **CHAIN** loads and runs Accounting.Section.Two

50: PRINT displays the new computed value of variable X, that is, 200 (X = X + 100).

NOTES

- The values of the variables left over from the previous program are not cleared.
- If an error is made, the following messages are displayed: ?FILE NOT FOUND ERROR, if the specified program in the **CHAIN** statement does not exist; ?REDIM ERROR, if the chained program dimensions an array that was dimensioned in the previous program.

CHR$

stands for **CHARACTER**

TYPE String function

FORMAT **CHR$** (*arithmetic expression*)

ACTION Converts an ASCII numeric code to its character equivalent.

ASCII stands for American Standard Code for Information Interchange. ASCII codes make up a table of standard numerical equivalents for a standard set of characters, called ASCII characters. ASCII characters include uppercase and lowercase letters, numbers, and special control and graphics characters. *arithmetic expression* is treated as an ASCII code (in decimal) and must be in the range from 0 (zero) to 255.

EXAMPLE

1. *arithmetic expression* can be a numeric constant;

   PRINT **CHR$** (65) Returns A
   PRINT **CHR$** (30 + 35) Returns A

2. a numeric variable;

   A = 65 : B = 30 : C = 35
   PRINT **CHR$** (A) Returns A
   PRINT **CHR$** (B + C) Returns A

3. any valid combination thereof.

   A = 2 : B = 10
   PRINT **CHR$** (A ^ 2 + B * 6 + 1) Returns A

NOTES

- If *arithmetic expression* is of the real type, Business BASIC will convert it to an integer.
- The ASCII numeric code for a capital A is 65.
- The ASC function is the inverse of the **CHR$** function. It converts a character back to its ASCII code.
- Business BASIC has 12 string or string-related functions: ASC, **CHR$**, HEX$, INSTR, LEFT$, LEN, MID$, RIGHT$, STR$, SUB$, TEN, VAL.

CLEAR

TYPE Statement

FORMAT **CLEAR**

ACTION Sets all numeric variables to 0 (zero) and all string variables to null.

EXAMPLE

```
10 A = 5 + 5 : B = 5 * 5 : A$ = "Before the CLEAR statement"
20 PRINT A,B
30 PRINT A$
40 CLEAR
50 PRINT A,B
60 PRINT A$
70 END
```

RESULT

Line 10: Assigns values to variables A and B, and string variable A$.

20: Prints the values of A and B: 10 25.

30: Prints the value of A$: Before the **CLEAR** statement.

40: Sets the variables A and B to zero and the string variable to null.

50: Prints the values of A and B: Ø Ø.

60: Prints the value of A$:

The result of line 60 is a blank line since a null string represents "no characters" and not a particular value.

NOTES

- If you want to "zero out" specific variables, use specific assignment statements rather than the **CLEAR** statement to avoid affecting the whole program.

  Example: A = Ø : A$ = ""

- The number of characters in a string expression may range from 0 (zero) to 255.
- A string variable is identified by a dollar sign ($).

CLOSE

TYPE File statement

FORMAT **CLOSE**

ACTION Causes all open devices and files to be closed.

A **CLOSE** statement with no *file number* specified causes all devices and files that have been opened to be closed. Closed files and devices must be reopened before they can be accessed again. The same or a different *file number* may be used.

EXAMPLE

```
10  OPEN#1, "Customers"
20  OPEN#3, "Statistics"
30  OPEN#5, ".Printer"
      |
70  CLOSE
80  END
```

NOTES

- **CLOSE** must always precede the END statement.
- All open files are closed when a LOAD, CLEAR, NEW, or RUN statement is executed. The CHAIN statement does not close any files.

CLOSE#

TYPE File statement

FORMAT **CLOSE#** *file number*

ACTION Closes the file whose reference number is specified after the number sign.

Closed files and devices must be reopened before they can be accessed again. The same or a different *file number* may be used.

EXAMPLE

```
 10 OPEN#1, "Customers"
 20 OPEN#3, "Statistics"
 30 OPEN#5, ".Printer"
      |
 70 CLOSE#1
 80 CLOSE#3
 90 CLOSE#5
100 END
```

NOTES

- **CLOSE#** must always precede the END statement.
- All open files are closed when a LOAD, CLEAR, NEW, or RUN statement is executed. The CHAIN statement does not close any files.

COLON

symbol **:**

TYPE Delimiter

FORMAT *statement* $\{$**:** *statement*$\}$

ACTION Separates statements in a list of statements or multiple statements written on the same line.

EXAMPLE

1. A = 1 **:** B = 2 **:** C = 3 **:** PRINT A,B,C
2. A$ = "AB" **:** B$ = "CD" **:** C$ = "EF" **:** PRINT A$ + B$ + C$
3. FOR X = 1 TO 3 **:** PRINT X **:** NEXT X
4. GOSUB 500 **:** GOSUB 750 **:** END
5. IF A = 1 THEN PRINT "WORKING" **:** GOSUB 1000 **:** PRINT "DONE"

RESULT

1. Three assignment statements and one print statement on a single line.
2. Three assignment statements and one print statement on a single line.
3. A FOR ... NEXT loop on a single line.
4. Two unconditional transfers to subroutines and an END statement that will be executed sequentially.
5. If A is not equal to 1, none of the statements in the list will be executed and the program will pass on to the next line; if A = 1 is true, all three statements in the list will be executed in turn.

NOTES

- Putting more than one statement on a single line saves memory space and speeds up program execution.

CONCATENATION *symbol* +

TYPE String operator

FORMAT *string expression* + *string expression*

ACTION Concatenates (joins together) two or more string expressions.

EXAMPLE

1. *string expression* can be a string constant;

PRINT "GOOD" + " MORNING"	Returns GOOD MORNING
PRINT "1234" + "567890"	Returns 1234567890
PRINT "A" + "B" + "C" + "D"	Returns ABCD

2. a string variable;

G$ = "GOOD" : M$ = " MORNING"	
PRINT G$ + M$	Returns GOOD MORNING

3. a substring function;

A$ = "ANOTHER"	
PRINT MID$ (A$,2,3)	Returns NOT

4. any valid combination thereof.

A$ = " AFTERNOON"	
PRINT "GOOD" + A$	Returns GOOD AFTERNOON

NOTES

- A blank space is also a character. A blank space has been inserted at the beginning of the strings: "AFTERNOON" and "MORNING".
- The number of characters in a string expression may range from 0 (zero) to 255. A null string is a string that contains no characters.

  Example: A$ = ""

  A null string is generally used to initialize string variables at the beginning of a program.

CONT

stands for **CONTINUE**

TYPE Statement

FORMAT **CONT**

ACTION Causes program execution to continue after a temporary break.

Program execution is temporarily halted by pressing CTRL-C, after a STOP or an END statement has been executed or an error has occurred. **CONT** is used to resume at the point where the break happened. Execution is resumed at the statement immediately following the STOP or END statement. If a program is halted by an error, execution is resumed with the statement in which the error occurred.

EXAMPLE

```
10  PRINT "THIS PROGRAM STARTS AT LINE NUMBER 10"
20  STOP : PRINT "EXECUTION CONTINUES WITH THIS PRINT
    STATEMENT"
```

RESULT

Line 10: Prints the string on the screen:

THIS PROGRAM STARTS AT LINE NUMBER 10

20: The STOP statement temporarily halts program execution and causes the following message to be displayed:

BREAK IN 20

(that is, in line 20).

Typing **CONT** on the keyboard and pressing the RETURN key cause execution to continue with the next instruction following the STOP statement at line 20.

20: Prints the string on the screen:

EXECUTION CONTINUES WITH THIS PRINT STATEMENT

NOTES

- You cannot use the **CONT** command after you add or alter statements in a program that has been halted by a STOP statement.

CONV

TYPE Numeric function

FORMAT **CONV** (*string expression* | *arithmetic expression*)

ACTION Evaluates the expression and returns a real value.

Numeric functions may be used either in immediate mode in conjunction with a PRINT statement or in deferred execution. If the argument is a *string,* then *string expression* must be a numeric string. All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions.

EXAMPLE

Print **CONV** (922337−203685)	Returns 718652
Print **CONV** (9223378−3036057)	Returns 6.18732E+06
Print **CONV** ("123456")	Returns 123456
Print **CONV** ("1234567")	Returns 1.23457E+06
Print **CONV** ("1234567.123")	Returns 1.23457E+06
Print **CONV** ("123.4567")	Returns 123.457

NOTES

- The value may be assigned to a regular integer. The conversion from real to integer is automatic in the latter case.
- If **CONV** is used with a string expression, the effect is the same as with the VAL function.

  Example:

  X = VAL ("1234567.123") : PRINT **CONV** (X)

  and

  PRINT **CONV** (VAL("1234567.123"))

  return the same value: 1.23457E+06.
- Beyond 6 digits, the value is expressed in exponential notation.
- Business BASIC has 16 numeric functions in the following type categories:

trigonometric:	ATN, COS, SIN, TAN
arithmetic:	ABS, EXP, INT, LOG, RND, SGN, SQR
conversion:	**CONV**, CONV%, CONV&, CONV$
user-defined:	DEF FN

CONV%

TYPE Numeric function

FORMAT **CONV%** (*arithmetic expression*)

ACTION Evaluates *arithmetic expression* and returns an integer value.

Numeric functions may be used either in immediate mode in conjunction with a PRINT statement or in deferred execution. The argument to all numeric functions must be an *arithmetic expression.* All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions.

EXAMPLE

PRINT **CONV%** (123.94)	Returns 124
PRINT **CONV%** (−123.94)	Returns −124

NOTES

- The returned integer value is rounded off to the nearest whole number.
- The percent sign (%) is an identifier that defines a function or a variable name as being of the integer type.
- The returned value by **CONV%** must be within the range from −32768 to 32767. Exceeding this range causes the ?OVERFLOW ERROR message to be displayed.
- Business BASIC has 16 numeric functions in the following type categories:

trigonometric:	ATN, COS, SIN, TAN
arithmetic:	ABS, EXP, INT, LOG, RND, SGN, SQR
conversion:	CONV, **CONV%**, CONV&, CONV$
user-defined:	DEF FN

CONV&

TYPE Numeric function

FORMAT **CONV&** (*string expression* | *arithmetic expression*)

ACTION Evaluates the expression and returns a long integer value.

Numeric functions may be used either in immediate mode in conjunction with a PRINT statement or in deferred execution. If the argument is a string, then *string expression* must be a numeric string. All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions.

EXAMPLE

```
10 PRINT CONV& (9876543210987654321-1234567890123456789)
20 PRINT CONV& ("1234567.123")
```

RESULT

Line 10: **CONV&** returns −3416920397562346125

20: **CONV&** converts the string expression into a numeric expression and extracts the integer portion of the value: 1234567 (no rounding off).

NOTES

- The ampersand (&) is an identifier that defines a function or a variable name as being of the long integer type.
- The value returned by the **CONV&** function must be within the range from −9223372036854775808 to 9223372036854775807. Exceeding this range would cause the ?OVERFLOW ERROR message to be displayed.
- If the expression is a string, the effect is the same as using the VAL function followed by **CONV&**.

  Example: X = VAL ("1234567.123") : PRINT **CONV&** (X)
- Business BASIC has 16 numeric functions in the following type categories:

trigonometric:	ATN, COS, SIN, TAN
arithmetic:	ABS, EXP, INT, LOG, RND, SGN, SQR
conversion:	CONV, CONV%, **CONV&**, CONV$
user-defined:	DEF FN

CONV$

TYPE Numeric function

FORMAT **CONV$** (*arithmetic expression*)

ACTION Evaluates *arithmetic expression* and returns a string value.

Numeric functions may be used either in immediate mode in conjunction with a PRINT statement or in deferred execution. The argument to all numeric functions must be an *arithmetic expression.* All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions.

EXAMPLE

```
10  A = 10203 : B = 20304 : T$ = CONV$ (A + B)
20  PRINT LEN (T$)
30  PRINT LEFT$ (T$,1)
40  PRINT MID$ (T$,2,3)
50  PRINT RIGHT$ (T$,1)
60  END
```

RESULT

Line 10: The evaluation of the numeric expression returns 30507. The numeric value 30507 is then converted into a string expression and assigned to the string variable T$.

20: PRINT LEN (T$)	Returns 5
30: PRINT LEFT$ (T$,1)	Returns 3
40: PRINT MID$ (T$,2,3)	Returns 050
50: PRINT RIGHT$ (T$,1)	Returns 7

NOTES

- A dollar sign ($) is an identifier that defines a function or a variable name as being of the string type.
- Business BASIC has 16 numeric functions in the following type categories:

trigonometric:	ATN, COS, SIN, TAN
arithmetic:	ABS, EXP, INT, LOG, RND, SGN, SQR
conversion:	CONV, CONV%, CONV&, **CONV$**
user-defined:	DEF FN

COS

stands for **COSINE**

TYPE Numeric function

FORMAT **COS** (*arithmetic expression*)

ACTION Returns the cosine of *arithmetic expression*.

Numeric functions may be used either in immediate mode in conjunction with a PRINT statement or in deferred execution. The argument to all numeric functions must be an *arithmetic expression.* All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions.

EXAMPLE

```
10  REM *** H = Hypotenuse of angle A
20  REM *** S = Side adjacent to angle A
30  REM *** A = Angle of a right triangle
40  FOR J = 1 TO 3
50  PRINT COS (J)
60  NEXT J
70  END
```

RESULT

Line 10–30: Remarks to document program.

40: Sets up a loop to repeat three times.

50: Prints the cosine of J:

.540302306 for J = 1 (radians)
–.416146836 for J = 2 (radians)
–.989992497 for J = 3 (radians)

60: Repeats from line 40.

NOTES

- ARCCOS is the opposite of **COS**. **COS** (A) = S/H *numeric expression* (expressed in Radians) is the angle whose cosine is to be calculated.
- *Conversions:*
  Radian = Degree / 57.29577951
  Degree = Radian * 57.29577951
- Business BASIC has 16 numeric functions in the following type categories:

trigonometric:	ATN, **COS**, SIN, TAN
arithmetic:	ABS, EXP, INT, LOG, RND, SGN, SQR
conversion:	CONV, CONV%, CONV&, CONV$
user-defined:	DEF FN

CREATE

TYPE Statement

FORMAT **CREATE** *pathname*, CATALOG|TEXT|DATA [, *arithmetic expression*]

ACTION Creates root directories, subdirectories, text files, and data files.

Program files are created with the SAVE command. CATALOG, TEXT, and DATA files are created with the **CREATE** statement. The type of a file is determined at the time the file is created, either by assignment with a **CREATE** statement or by the first access method used after creating the file with a OPEN# statement.

EXAMPLE

```
10 CREATE "Memories/Part.One", TEXT, 4096
```

COMMENTS

- *pathname* must be enclosed in quotation marks. Quotation marks may be omitted only in immediate mode.
- The volume name and the local name must be preceded with a slash (/). The slash may be omitted if the prefix has been set to Memories. The complete pathname is thus assumed to be the contents of the reserved variable PREFIX$ plus the partial pathname as entered after **CREATE**.
- A comma must separate the pathname from the type of the file.
- A file record size defaults to 512 bytes. The record size is required only for random-access files and must be specified by any positive *arithmetic expression* following the file type.

NOTES

- The type of file is specified by the following reserved words:

CATALOG	For directories or subdirectories files
TEXT	For text files
DATA	For data files

- To change the type of a file, you must first delete it and then recreate it.

DATA

TYPE Statement

FORMAT **DATA** *constant* {[, *constant*]}

ACTION Contains *constants* that are accessed by one or more READ statements.

constant may be numeric (real, integer, or long integer), or alphanumeric (string or literal).

You can put as many constants in a list of constants as will fit on a line. A **DATA** statement is not executable by itself; a READ statement is used to accept each data item and assign it sequentially to corresponding variables. The variable type of the READ statement must match the corresponding constant type in the **DATA** statement. The information contained in multiple **DATA** statements is read as if it were one continuous list. The READ statements access the **DATA** statements in line number order.

EXAMPLE

```
10  FOR D = 1 TO 3
20  READ X
30  PRINT X
40  NEXT D
50  DATA 10, 20, 30
```

RESULT

Line 10: Sets up a loop to repeat three times.

20: Reads the next item in **DATA** list and assigns it to the variable X.

30: Prints X.

40: Repeats from line 10.

50: Contains three data items.

NOTES

- String constants in **DATA** statements do not need to be surrounded by quotation marks unless the string contains commas, colons, or blanks.
- **DATA** statements may be placed anywhere in the program.

DEF FN *stands for* **DEFINITION** *and* **FUNCTION**

TYPE User-defined statement

FORMAT **DEF FN** *function name* (*real variable*) = *arithmetic expression*

ACTION Defines a user-created function.

real variable is a dummy variable used in *arithmetic expression* to define a function. The resulting function can be used in other expressions or statements when the function is called by its name.

EXAMPLE

```
10 DEF FNA (X) = INT (X * 100 + .5)/100
20 DEF FNB (X) = INT (X * 1000 + .5)/1000
30 M = 6.123456
40 PRINT FNA (M)
50 PRINT FNB (M)
60 END
```

RESULT

Line 10: Definition of function A for rounding off to 2 decimals.

20: Definition of function B for rounding off to 3 decimals.

30: Assignment of the value 6.123456 to the variable M.

40: Prints the value M with 2 decimals (user-defined function **FN**A).

50: Prints the value M with 3 decimals (user-defined function **FN**B).

NOTES

- The dummy variables (X in the example) serve to define the function. By themselves they have no effect on the output value and do not become reserved variables for the program as a whole.
- After the definition of the function, any numeric constant, numeric variable, or arithmetic expression can be substituted for the "dummy variables" in parentheses.

DEL

stands for **DELETE**

TYPE Statement

FORMAT **DEL** *line number1* [TO |,|– *line number2*]

ACTION Deletes one or more specified program lines.

EXAMPLE

1. **DEL** 10
2. **DEL**10–50
3. **DEL** –50
4. **DEL** 50–
5. **DEL** 10, 50–100

RESULT

1. Deletes line 10.
2. Deletes all lines numbered from 10 to 50 inclusive.
3. Deletes all lines from the beginning of the program until line 50 inclusive.
4. Deletes all lines from line 50 to the end of the program.
5. Deletes line 10 and all lines numbered from 50 to 100 inclusive.

NOTES

- To delete a single line, type the line number and press the RETURN or ENTER key.
- The NEW command deletes the entire program.

DELETE

TYPE File statement

FORMAT **DELETE** *pathname*

ACTION Deletes a file from the disk.

The **DELETE** statement deletes local files, root directories, and subdirectories. A subdirectory may be removed only if all files in that directory have been deleted. If the last file in a root directory is deleted, the empty root directory will still remain.

EXAMPLE

DELETE/Stock/Purchases/France

RESULT

The file named France will be deleted, but the empty root directory named Purchases will still remain.

NOTES

- Errors that can occur with nonvalid **DELETE** statements are:

Cause	*Error Message*	*Code*
One or more files are open	?FILES BUSY ERROR	23
Disk is write-protected	?WRITE PROTECTED ERROR	27
Nonexistent local file name	?FILE NOT FOUND ERROR	30
Nonexistent subdirectory	?PATH NOT FOUND ERROR	31
Nonexistent volume name	?VOLUME NOT FOUND ERROR	32
Specified file is locked Subdirectory contains files	?FILE LOCKED ERROR	35

DIM

stands for **DIMENSION**

TYPE Statement

FORMAT **DIM** *variable name* (*subscripts*) {[, *variable* (*subscripts*)]}

ACTION Allocates memory storage for arrays by setting the maximum values for variable subscripts.

An array is a set or matrix of variables identified by subscripts. *subscripts* is a list of numeric expressions, separated by commas, which defines the dimensions of the array. When executed, the **DIM** statement sets the numeric array's elements to an initial value of 0 (zero) and the string array's elements to an initial null value. An array variable can have more than one subscript, defining a multidimensional array.

EXAMPLE

```
10 DIM AR (4, 3)
20 FOR X = 1 TO 4
30 FOR Y = 1 TO 3
40 READ AR (X,Y)
50 NEXT Y
60 NEXT X
70 DATA 1,2,3,4,5,6,7,8,9,10,11,12
80 END
```

RESULT

Line 10: Specifies memory storage to be allocated to the 12 elements of array AR (4 × 3 = 12).

20: Sets up a loop for the 4 rows of the array.

30: Sets up a loop for the 3 columns of the array.

40: Reads and assigns the 12 values of the DATA statement to the 12 elements of the array.

50: Repeats from line 30.

60: Repeats from line 20.

70: Contains 12 data items.

NOTES

- If an array variable name is not defined by a **DIM** statement, BASIC automatically reserves a default size of 11 elements.
- A subscript's minimum value is always 0 (zero). **DIM** A(4) dimensions a list with four elements: A(0), A(1), A(2), A(3).

DIV

stands for **INTEGER DIVISION**

TYPE Arithmetic operator

FORMAT *arithmetic expression1* **DIV** *arithmetic expression2*

ACTION Evaluates the integer result of a division.

EXAMPLE

A& = 7 : B& = 2 : PRINT A& **DIV** B& Returns 3

NOTES

- Operands of **DIV** can only be long integers.
- Business BASIC has 9 arithmetic operators:

+	Unary plus
−	Unary minus
^	Exponentiation
*	Multiplication
/	Floating-point division
MOD	Modulo division
DIV	Integer division
+	Addition
−	Subtraction

DIVISION

symbol /

TYPE Arithmetic operator

FORMAT *numeric expression1 / numeric expression2*

ACTION Performs an arithmetic division.

EXAMPLE

1. *numeric expression* can be a numeric constant;

PRINT 20 / 10	Returns 2
PRINT 40 / 10 / 2	Returns 2
PRINT 20 / (−10)	Returns 2

2. a numeric variable;

A = 40 : B = 20 : C = 10 : D = 2 : E = −10	
PRINT B / C	Returns 2
PRINT A / C / D	Returns 2
PRINT B / (−E)	Returns 2

3. any valid combination thereof.

A = 40 : B = 20 : C = 10 : D = 2 : E = −10	
PRINT 20 / C	Returns 2
PRINT A / C / D	Returns 2
PRINT 20 / (−E)	Returns 2

NOTES

- Business BASIC has 9 arithmetic operators:

+	Unary plus
−	Unary minus
^	Exponentiation
*	Multiplication
/	Floating-point division
MOD	Modulo division
DIV	Integer division
+	Addition
−	Subtraction

DOLLAR

symbol **$**

TYPE Identifier

FORMAT *string variable name***$**

ACTION Identifies the variable as being of the string type.

Variables have identifiers attached to specify which type of value they represent.

EXAMPLE

```
10 A$ = "THE $ IDENTIFIER "
20 A1$ = "DEFINES A VARIABLE "
30 AA$ = "AS BEING OF "
40 ALPHA$ = "THE STRING TYPE"
50 PRINT A$ + A1$ + AA$ + ALPHA$
60 END
```

RESULT

Line 10–40: Four assignment statements of strings to string variables.

50: Prints the four strings concatenated (joined together) into one string:

THE $ IDENTIFIER DEFINES A VARIABLE AS BEING OF THE STRING TYPE.

NOTES

- The use of a reserved word as a variable is illegal:

  Example: CHR**$**

- The number of characters in a string expression may range from 0 (zero) to 255. A null string is a string that contains no characters.

  Example: A**$** = ""

- A numeric variable without an identifier is automatically of the single-precision type.
- Business BASIC has three identifiers attached to variable names:

  & For variables of the long integer type
  % For variables of the integer type
  $ For variables of the string type

E — *stands for* EXPONENTIAL NOTATION

TYPE Operator

FORMAT *number* **E** *positive* or *negative exponent*

ACTION Indicates exponential (or scientific) notation.

The letter **E** means "times 10 to the power of the exponent." Any real number can be expressed in exponential notation, which is particularly useful for very large numbers or small fractions.

EXAMPLE

1. 1234**E**−2	Exponential notation for 12.34
2. 0.1234**E**2	Exponential notation for 12.34
3. 0.1234**E**+2	Exponential notation for 12.34

NOTES

- A positive exponent is assumed if no sign is used.
- With a plus sign (+), the decimal point is moved to the right. With a minus sign (−), the decimal point is moved to the left. The number of places is indicated by the number following the letter **E**.
- Business BASIC has 9 arithmetic operators:

+	Unary plus
−	Unary minus
^	Exponentiation
*	Multiplication
/	Floating-point division
MOD	Modulo division
DIV	Integer division
+	Addition
−	Subtraction

ELSE

TYPE Statement

FORMAT : **ELSE** [*arithmetic expression* | *line number*]

ACTION If the condition of an IF ... THEN statement is true, the statement list following THEN is executed. If the condition is false, the statement list that follows **ELSE** is executed instead.

EXAMPLE

```
10  INPUT X%
20  IF X% = 1 THEN GOSUB 1000 : ELSE GOSUB 2000
```

RESULT

Line 10: Accepts input and assigns it to the variable X%.

20: If X% = 1, sends program to line 1000; otherwise (if X% <> 1), sends program to line 2000.

NOTES

- Business BASIC has 6 relational operators:

=	Equal to
<> *or* ><	Not equal to
>	Greater than
>= *or* =>	Greater than or equal to
<	Less than
<= *or* =<	Less than or equal to

END

TYPE Statement

FORMAT **END**

ACTION Marks the end of a program or subroutine.

Terminates program execution, closes all files, and returns to command (keyboard) level. **END** statements may be placed anywhere in the program.

EXAMPLE

```
  10 INPUT Q$
  20 IF Q$ = "YES" THEN 1000
     |
1000 END
```

```
  10 INPUT Q$
  20 IF Q$ = "YES" THEN END
```

```
  10 GOSUB 1000
  20 END
     |
1000 PRINT "SUBROUTINE"
1010 RETURN
```

NOTES

- After an **END** statement is executed, BASIC always returns to command level. **END** at the end of a program is optional.
- STOP also terminates program execution. However, STOP displays a "Break" message, whereas **END** does not, and STOP does not automatically close files.

EOF

stands for **END OF FILE**

TYPE File reserved variable

FORMAT **EOF**

ACTION Contains the reference number of the file causing an end-of-file error.

EXAMPLE

1. PRINT **EOF**
2. ON (**EOF**) GOTO 1000, 2000, 3000

RESULT

1. Determines the file that has caused an end-of-file error.
2. Program execution branches to line numbers 1000, 2000, or 3000 according to the value assigned to the variable **EOF**.

NOTES

- When used with conditional statements, **EOF** must be enclosed in parentheses.

EQUAL TO

symbol =

TYPE Relational operator

FORMAT *expression1* = *expression2*

ACTION Allows a logical comparison to be made between two expressions.

expression1 and *expression2* are either both numeric or both string. The comparison returns a logical value. If both expressions have equivalent values, the result of the comparison is true (non-zero, represented by the numerical value −1); otherwise, the expression is false (zero, represented by 0). Relational operators are usually used in conditional statements, such as IF ... THEN statements, to make a decision regarding program flow.

EXAMPLE

```
10 A = 10 : B = 20 : C = 2 : X$ = "TRUSTY" : Y$ = "TRUST"
20 IF A = B THEN PRINT "TRUE" : ELSE PRINT "FALSE"
30 IF A = B/C THEN PRINT "TRUE" : ELSE PRINT "FALSE"
40 IF X$ > = Y$ THEN PRINT "TRUE" : ELSE PRINT "FALSE"
```

RESULT

Line 10: Assigns values to the numeric variables A, B, C, and the string variables X$ and Y$.

20: Since A is not equal to B, prints: FALSE.

30: Since A is equal to B divided by C, prints: TRUE.

40: Since TRUSTY is not equal to TRUST, prints: FALSE.

NOTES

- The strings are compared character by character, from left to right, on the basis of their ASCII code numbers. The first character found in one string that has a greater ASCII value than the character found in the same position in the second string makes the first string greater. If the characters in the same positions are identical but one string's current length is longer, the longer string is greater.
- Business BASIC has 6 relational operators:

=	Equal to
<> *or* ><	Not equal to
>	Greater than
>= *or* =>	Greater than or equal to
<	Less than
<= *or* =<	Less than or equal to

ERR

stands for **ERROR**

TYPE File reserved variable

FORMAT **ERR**

ACTION Contains the code number corresponding to the type of the detected error.

EXAMPLE

```
10  ON ERR GOTO 70
20  DIM A (12)
30  FOR X = 1 TO 12 : READ A : NEXT X
40  GOTO 80
50  DATA 1, 2, 3, 4, 5, 6
60  END
70  IF ERR = 4 THEN RESUME 40
80  PRINT "Program execution continues"
```

RESULT

Line 10: If an error occurs, ON **ERR** causes an unconditional branching to line number 70.

20: Dimensions a 12-element list.

50: Since the DATA statement contains only 6 data items, the unconditional branching ON **ERR** GOTO 70 is executed.

70: Program execution resumes at line 40 (the code number of the ?OUT OF DATA ERROR is 4).

40: GOTO causes an unconditional branching to line 80.

80: Program execution continues at line 80.

NOTES

- **ERR** is usually used in IF ... THEN conditional statements to direct program flow to the error-handling subroutines.
- You can refer to **ERR** to determine what kind of error occurred.

ERRLIN *stands for* **ERROR** *and* **LINE**

TYPE File reserved variable

FORMAT **ERRLIN**

ACTION Contains the line number where an error occurred.

EXAMPLE

```
 10  ON ERR GOTO 100
 20  INPUT N
 30  IF N = 9 THEN END
 40  A = 12/N
 50  PRINT A : GOTO 20
100  N = N + 1
110  PRINT "Error at line number"; ERRLIN
120  PRINT "Error code number "; ERR
130  RESUME
140  END
```

RESULT

Line 10: If an error occurs, the ON ERR statement causes program execution to branch at line 100.

20: A division by zero is considered and "error." If a 0 (zero) is input and assigned to variable N, program execution automatically branches to line 100.

100: Entry point of the error-handling subroutine. N is reinitialized: (Ø + 1 = 1).

110: Displays the line number where the error occurred.

120: Displays the code number of the error (14 for a DIVISION BY ZERO error).

130: Causes program execution to branch again to line 40.

40: Variable A is assigned the result of the new computation: A = 12/1.

50: Displays the result, and program continues at line 20.

NOTES

- **ERRLIN** is usually used in IF ... THEN conditional statements to direct program flow to the error-handling subroutines.

EXEC

stands for **EXECUTE**

TYPE File statement

FORMAT **EXEC** *pathname*

ACTION Starts sequential execution automatically directed by programs stored in a *text file*.

EXAMPLE

EXEC ".D2/PILOT"

RESULT

Assuming:

1. we had previously saved on disk three programs named PROGRAM.ONE, PROGRAM.TWO, and PROGRAM.THREE, respectively;
2. there also is on disk a text file named PILOT containing the statements: RUN PROGRAM.1, RUN PROGRAM.2, and RUN PROGRAM.3.

The **EXEC** command will direct automatic and sequential execution of the three programs by reading the contents of the PILOT file and acting on this as though you were typing the same commands from the keyboard.

NOTES

- After the three programs' execution is terminated, control is returned to the keyboard. Control is also returned to the keyboard if:

  —program execution is stopped by pressing CONTROL-C
  —a STOP statement is encountered
  —an error occurs
  —an *end-of-file* marker is encountered

- If an INPUT or a GET statement occurs in a program, it takes its input from the next line of the text file, *not* the keyboard.
- **EXEC** automatically opens the file it uses.

EXP

stands for **EXPONENTIATION**

TYPE Numeric function

FORMAT **EXP** (*arithmetic expression*)

ACTION Returns the value of E to the power of *arithmetic expression*.

The mathematical number *e* (2.718289) is the base for natural logarithms. Numeric functions may be used either in immediate mode in conjunction with a PRINT statement or in deferred execution. The argument to all numeric functions must be an *arithmetic expression.* All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions.

EXAMPLE

1. *arithmetic expression* can be a numeric constant:

PRINT **EXP** (2)	Returns 7.3890561
PRINT **EXP** (6−2)	Returns 54.5981501

2. a numeric variable;

A = 2 : B = 6	
PRINT **EXP** (A)	Returns 7.3890561
PRINT **EXP** (B−A)	Returns 54.5981501

3. any valid combination thereof.

10 FOR J = 2 TO 6 STEP 2	Returns 7.3890561
20 PRINT **EXP** (J)	Returns 54.5981501
30 NEXT J	Returns 403.428793

NOTES

- **EXP** is the opposite of LOG.

  Example: E = **EXP** (2) : L = LOG (E) : PRINT L   Returns 2

- Business BASIC has 16 numeric functions in the following type categories:

trigonometric:	ATN, COS, SIN, TAN
arithmetic:	ABS, **EXP**, INT, LOG, RND, SGN, SQR
conversion:	CONV, CONV%, CONV&, CONV$
user-defined:	DEF FN

EXPONENTIATION *symbol* ^

TYPE Arithmetic operator

FORMAT *base* ^ *power*

ACTION Performs an arithmetic exponentiation, that is, raises *base* to the power of *power*.

base and *power* are both *numeric expressions.*

EXAMPLE

1. *numeric expression* can be a numeric constant;

PRINT 10 ^ 2	Returns 100
PRINT 10 ^ 2 ^ 2	Returns 10000
PRINT −10 ^ (−2)	Returns −.01

2. a numeric variable;

B = 10 : P = 2	
PRINT B ^ P	Returns 100
PRINT B ^ P ^ P	Returns 10000
PRINT −B ^ (−P)	Returns −.01

3. any valid combination thereof.

B = 10 : P = 2	
PRINT B ^ 2	Returns 100
PRINT 20 ^ P ^ 2	Returns 10000
PRINT −10 ^ (−P)	Returns −.01

NOTES

- In the example, the base = 10 and the power = 2.
- Business BASIC has 9 arithmetic operators:

+	Unary plus
−	Unary minus
^	Exponentiation
*	Multiplication
/	Floating-point division
MOD	Modulo division
DIV	Integer division
+	Addition
−	Subtraction

FN

stands for **FUNCTION**

TYPE User-defined function

FORMAT **FN** *function name* (*arithmetic expression* {[, *arithmetic expression*]}

ACTION Processes the value given by *arithmetic expression* according to a previously defined set of operations.

The DEF **FN** statement is used to define a function as a particular set of operations and to give the function a name (beginning with **FN**). User-defined functions serve the same purposes as predefined built-in functions.

EXAMPLE

```
10  DEF FNA (X) = INT (X * 100 + .5)/100
20  DEF FNB (X) = INT (X * 1000 + .5)/1000
30  M = 6.123456
40  PRINT FNA (M)
50  PRINT FNB (M)
60  END
```

RESULT

Line 10: Definition of function A for rounding off to 2 decimals.

20: Definition of function B for rounding off to 3 decimals.

30: Assignment of the value 6.123456 to the variable M.

40: Prints the value M with 2 decimals (user-defined function **FN**A).

50: Prints the value M with 3 decimals (user-defined function **FN**B).

NOTES

- The variable X, enclosed in parentheses after the keyword **FN** in the DEF statement, is called a dummy variable; it is used again in the operation to the right of the equal sign, in order to define the relationships. Using the variable X in this way has no effect on the program as a whole or on the value of X used in any other context within the program. After the definition of the function, any numeric constant, numeric variable, or arithmetic expression can be substituted for the dummy variable X in parentheses.

FOR

TYPE Statement

FORMAT **FOR** *control variable* = *aexpr1* TO *aexpr2* [STEP *aexpr3*]

|

NEXT [*control variable* {, *control variable* }]

ACTION Sets up a program loop that repeats the series of instructions inside the loop a given number of times.

aexpr is an *arithmetic expression.* The loop begins with the **FOR** statement and ends with the NEXT statement. Every instruction in between is executed once with each repetition. Every repetition automatically increments (adds to) the value of *control variable* by a value equal to *expr3*; if STEP is omitted, the default increment is 1. *control variable* starts off having a value equal to *expr1*; when the value of *control variable* reaches *expr2*, the loop is ended and program execution continues with the statement after NEXT. A conditional statement can be used to exit the loop before it is finished.

EXAMPLE

```
10  FOR B = 1 TO 10
20  PRINT "AZ";
30  NEXT B
40  END
```

RESULT

Line 10: Sets up a loop to repeat 10 times.

20: Prints string AZ.

30: Repeats from line 10.

NOTES

- The initial value of *control variable* B has been incremented by the default value of 1.
- A loop structure may contain other loops within it, provided that the loops are nested.

FRE *stands for* **FREE**

TYPE Statement

FORMAT **FRE**

ACTION Returns the number of bytes of memory remaining available to the user.

EXAMPLE

```
10 IF FRE < 6000 THEN 30
20 PRINT "Sufficient memory available" : END
30 PRINT "Insufficient free memory".
40 END
```

RESULT

Line 10: If there are fewer than 6000 bytes of free memory, program execution jumps to line 30; otherwise, it defaults to line 20.

20: Prints message: "Sufficient memory available" if there are 6000 or more bytes of free memory available.

30: Prints message: "Insufficient free memory".

NOTES

- Whenever possible, the use of:

  —multiple line statements
  —no REM statements
  —integer array variables
  —variables instead of constants
  —GOSUB statements
  —0 (zero) elements of matrices

  will save memory space and speed up program execution.

GET

TYPE Statement

FORMAT **GET** *variable*

ACTION Gets a single character from the keyboard and assigns it to *variable*. The character is not displayed on the screen, and the user is not required to press the RETURN key.

EXAMPLE

```
  10 PRINT "Type C to Continue, E to End."
  20 GET C$
  30 IF C$ = "C" THEN GOSUB 1000 : END
  40 If C$ = "E" THEN END
  50 PRINT "Invalid entry. Try again." : GOTO 10
  60 END
     :::::::::::::::::::::::::::::::::::::::::::::::::::::::::
1000 REM *** SUBROUTINE
     |
2000 RETURN
```

RESULT

Line 10: Prints the message.

20: Returns any character entered at the keyboard as C$.

30: If C is typed, program execution passes to the subroutine at line 1000.

40: If E is typed, ends the program.

50: If any other character is typed, prints the message and jumps back to line 10.

60: Ends the program.

1000: Start of the subroutine.

2000: Returns the program execution to the next statement following the most recently executed GOSUB statement.

NOTES

- The **GET** statement may be followed by either a numeric or an alphanumeric variable. However, there are restrictions on entries if the variable is defined as numeric, and most programmers assign the input to a string variable and then convert the string to a number using the VAL function.

GOSUB

stands for **GO** *and* **SUBROUTINE**

TYPE Statement

FORMAT **GOSUB** *line number*

|

RETURN

ACTION Transfers program execution unconditionally to *line number*.

GOSUB is used to set up subroutines that can be used more than once by various parts of the program. *line number* is the first line of the subroutine. The subroutine consists of the statements between *line number* and RETURN. The RETURN statement causes program execution to continue with the next executable statement after **GOSUB**.

EXAMPLE

```
10  PRINT "Type C to Continue, E to End."
20  INPUT C$
30  IF C$ = "C" THEN GOSUB 1000 : END
40  IF C$ = "E" THEN END
50  PRINT "Invalid entry. Try again." : GOTO 10
60  END
    : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : :
1000 REM *** SUBROUTINE
     |
2000 RETURN
```

RESULT

Line 10: Prints the message.

20: Accepts input and assigns it to variable the C$.

30: If C is typed, program execution passes to the subroutine at line 1000.

40: If E is typed, ends the program.

50: If any other character is typed, prints the message and jumps back to line 10.

60: Ends the program.

1000: Start of the subroutine.

2000: Returns the program execution to the next statement following the most recently executed **GOSUB** statement.

NOTES

- A subroutine must always end with a RETURN statement to cause program execution to continue from the next statement following the **GOSUB** statement.

GOTO

TYPE Statement

FORMAT **GOTO** *line number*

ACTION Transfers program execution unconditionally to a specified line number.

EXAMPLE

```
10 GOTO 40
20 PRINT "PROGRAM EXECUTION JUMPED BACK TO LINE 20"
30 END
40 PRINT "PROGRAM EXECUTION IS TRANSFERRED TO LINE 40"
50 GOTO 20
```

RESULT

Line 10: Program execution is transferred to line 40.

40: Prints the string:

PROGRAM EXECUTION IS TRANSFERRED TO LINE 40

50: Program execution returns to line 20.

20: Prints the string:

PROGRAM EXECUTION JUMPED BACK TO LINE 20

30: END of the program.

NOTES

- If *line number* refers to a nonexecutable statement (such as REM or DATA), program execution continues with the first executable statement encountered at the next higher line number.
- In debugging, the **GOTO** statement can be used in direct mode to resume execution from a desired point in the program.

GREATER THAN *symbol* >

TYPE Relational operator

FORMAT *expression1* > *expression2*

ACTION Allows a logical comparison to be made between two expressions.

expression1 and *expression2* are either both numeric or both string. The comparison returns a logical value. If *expression1* has a greater value than *expression2*, the result of the comparison is true (non-zero, represented by the numerical value −1); otherwise, the result is false (represented by 0). Relational operators are usually used in conditional statements, such as IF ... THEN statements, to make a decision regarding program flow.

EXAMPLE

```
10 A = 10 : B = 20 : C = 2 : X$ = "TRUSTY" : Y$ = "TRUST"
20 IF B > A THEN PRINT "TRUE" : ELSE PRINT "FALSE"
30 IF A > B/C THEN PRINT "TRUE" : ELSE PRINT "FALSE"
40 IF X$ > Y$ THEN PRINT "TRUE" : ELSE PRINT "FALSE"
```

RESULT

Line 10: Assigns values to the numeric variables A, B, C, and the string variables X$ and Y$.

20: Since B is greater than A, prints: TRUE.

30: Since A is not greater than B divided by C, prints: FALSE.

40: Since TRUSTY is greater than TRUST, prints: TRUE.

NOTES

- The strings are compared character by character, from left to right, on the basis of their ASCII code numbers. The first character found in one string that has a greater ASCII value than the character found in the same position in the second string makes the first string greater. If the characters in the same positions are identical but one string's current length is longer, the longer string is greater.
- Business BASIC has 6 relational operators:

=	Equal to
<> *or* ><	Not equal to
>	Greater than
>= *or* =>	Greater than or equal to
<	Less than
<= *or* =<	Less than or equal to

GREATER THAN OR EQUAL TO >= *or* =>

TYPE Relational operator

FORMAT *expression1* >= *expression2*

ACTION Allows a logical comparison to be made between two expressions.

expression1 and *expression2* are either both numeric or both string. The comparison returns a logical value. If the value of *expression1* is greater than or equivalent to *expression2*, the result of the comparison is true (non-zero, represented by the numerical value −1); otherwise, the result is false (zero, represented by 0). Relational operators are usually used in conditional statements, such as IF ... THEN statements, to make a decision regarding program flow.

EXAMPLE

```
10 A = 10 : B = 20 : C = 2 : X$ = "TRUSTY" : Y$ = "TRUST"
20 IF B>= A THEN PRINT "TRUE" : ELSE PRINT "FALSE"
30 IF A>= B/C THEN PRINT "TRUE" : ELSE PRINT "FALSE"
40 IF X$>= Y$ THEN PRINT "TRUE" : ELSE PRINT "FALSE"
```

RESULT

Line 10: Assigns values to the numeric variables A, B, C, and the string variables X$ and Y$.

20: Since B is greater than A, prints: TRUE.

30: Since A is equal to B divided by C, prints: TRUE.

40: Since TRUSTY is greater than TRUST, prints: TRUE.

NOTES

- The strings are compared character by character, from left to right, on the basis of their ASCII code numbers. The first character found in one string that has a greater ASCII value than the character found in the same position in the second string makes the first string greater. If the characters in the same positions are identical but one string's current length is longer, the longer string is greater.
- Business BASIC has 6 relational operators:

=	Equal to
<> *or* ><	Not equal to
>	Greater than
>= *or* =>	Greater than or equal to
<	Less than
<= *or* =<	Less than or equal to

HEX$

stands for **HEXADECIMAL**

TYPE String function

FORMAT **HEX$** (*arithmetic expression*)

ACTION Returns a string that represents the hexadecimal value of *arithmetic expression.*

EXAMPLE

```
10  FOR J = 1 TO 15
20  PRINT HEX$(J)
30  NEXT J
```

RESULT

Line 10: Sets up a loop to repeat 15 times.

20: Displays the hexadecimal value of the decimal value of variable J:

1	2	3	4	5	6
7	8	9	A	B	C
D	E	F			

30: Repeats from line 10.

NOTES

- The dollar sign ($) is an identifier that defines a function or a variable name as being of the string type.
- *arithmetic expression* is rounded to an integer before it is evaluated. For instance, 15.36 would be rounded to 15 before the equivalent hexadecimal value (F) is returned.
- *arithmetic expression* must be in the decimal range from −65535 to +65535. If *arithmetic expression* is negative, the two's complement form is used, that is,

  HEX$** (−*expression*) = **HEX$ (65535−*expression*)

  Both A$ = **HEX$** (−25) and B$ = **HEX$** (65536−25) return FFE7.
- Business BASIC has 12 string or string-related functions: ASC, CHR$, **HEX$**, INSTR, LEFT$, LEN, MID$, RIGHT$, STR$, SUB$, TEN, VAL.

HOME

TYPE Statement

FORMAT **HOME**

ACTION Clears the screen and sets the cursor to the upper-leftmost position.

EXAMPLE

```
10 HOME
20 INVERSE
30 PRINT "BLACK characters on a WHITE background"
40 FOR T = 1 TO 1000 : NEXT T
50 NORMAL
60 PRINT "WHITE characters on a BLACK background"
70 END
```

RESULT

Line 10: Clears the screen and sets the cursor to the upper-leftmost position.

20: Sets the inverse display mode.

30: Displays the string:

BLACK characters on a WHITE background

40: Delay loop.

50: Restores the normal display mode.

60: Displays the string:

WHITE characters on a BLACK background

NOTES

- No parameter is required after **HOME**.
- **HOME** may be used either in the immediate (command) mode by typing **HOME** and pressing the RETURN or ENTER key or in the deferred (program) mode with a line number.

HPOS *stands for* **HORIZONTAL** *and* **POSITION**

TYPE Reserved variable

FORMAT **HPOS** = *arithmetic expression*

ACTION Specifies the horizontal position of the cursor within a "window" or the total screen.

You can find the current position of the cursor by referring to the value of **HPOS** in a PRINT command/statement. The current horizontal position is relative to the left margin of the window. *arithmetic expression* can be any integer constant or variable or any real arithmetic expression.

EXAMPLE

HPOS = 6
moves the cursor horizontally to the sixth column within the current window.

NOTES

- All parameters are relative to the current window dimensions. For instance, in **HPOS** = 1, 1 specifies the first column within the current window.
- When **HPOS** is used to move the cursor horizontally, the cursor's vertical position is not affected.
- Values must be within the range from 0 (zero) to 255. A value of 0 (zero) is automatically converted to a value of 1. **HPOS** cannot move the cursor to a position outside the window. **HPOS** values greater than the width of the window cause the cursor to move to the righthand margin of the window.

IF ... GOTO

TYPE Statement

FORMAT **IF** *logical expression* **GOTO** *line number* [:ELSE *line number* | *statement list*]

ACTION Sends program execution to *line number* if *logical expression* is true (non-zero); otherwise:

1. if no ELSE clause is used, program execution passes to the next line in sequence;
2. if an ELSE clause is used, program execution passes to *line number* or *statement list* following ELSE.

IF ... GOTO is called a conditional statement; it is one of the most commonly used statements in BASIC. It redirects program execution on the basis of the truth or falsity of *logical expression*. *logical expression* is usually a relational expression, comparing two values with relational operators.

EXAMPLE

```
10 INPUT "YES OR NO"; X$
20 IF X$ = "YES" GOTO 40
30 IF X$ = "NO" GOTO 50 : ELSE 10
40 PRINT "Program execution is transferred to line 40" : END
50 PRINT "Program execution is transferred to line 50"
```

RESULT

Line 10: Asks for input and assigns it to variable X$.

20: If X$ is YES, program execution jumps to line 40.

30: If X$ is NO, execution jumps to line 50; otherwise, the statement following ELSE is executed.

40: Prints the message. Ends the program.

50: Prints the message.

NOTES

- The ELSE clause cannot be on a separate program line.

IF ... THEN

TYPE Statement

FORMAT **IF** *logical expression* **THEN** *line number* [:ELSE *line number* | *statement list*]

ACTION Sends program execution to *line number* or executes *statement list* following **THEN** if *logical expression* is true (non-zero); otherwise:

1. if no ELSE clause is used, program execution passes to the next line in sequence;
2. if the ELSE clause is used, program execution passes to *line number* or *statement list* following ELSE.

IF ... THEN is called a conditional statement; it is one of the most commonly used statements in BASIC. It redirects program execution on the basis of the truth or falsity of *logical expression*. *logical expression* is usually a relational expression, comparing two values with relational operators.

EXAMPLE

```
10 INPUT "YES OR NO"; X$
20 IF X$ = "YES" THEN 40
30 IF X$ = "NO" THEN 50 : ELSE 10
40 PRINT "Program execution is transferred to line 40" : END
50 PRINT "Program execution is transferred to line 50"
```

RESULT

Line 10: Asks for input and assigns it to variable X$.

20: If X$ is YES, program execution jumps to line 40.

30: If X$ is NO, execution jumps to line 50; otherwise, the statement following ELSE is executed.

40: Prints the message. Ends the program.

50: Prints the message.

NOTES

- The ELSE clause cannot be on a separate program line.

IMAGE

TYPE Statement

FORMAT **IMAGE** *specification* [{, *specification*}]

ACTION Stores the format specifications to be used by a PRINT [#] USING statement.

The PRINT USING and PRINT# USING statements are collectively referred to as PRINT [#] USING. PRINT [#] USING must refer to the line number of the **IMAGE** statement that it uses to format output.

EXAMPLE

```
10 IMAGE +###.###
20 PRINT USING 10; 1.1, 1.23456. 123.1, 1
```

RESULT

Line 10: Format specification.

20: PRINT USING displays output according to the format specified by the **IMAGE** statement at line 10:

```
  +1.100
  +1.235
+123.100
  +1.000
```

NOTES

- **IMAGE** can only be used as a program statement.
- Each specification, separated by a comma, corresponds to one printing field and controls the displayed format of the corresponding value.
- A single format specification may be used to display more than one numeric value.

INDENT

TYPE Reserved variable

FORMAT **INDENT** = *arithmetic expression*

ACTION Contains the number of spaces to be used to indent FOR ... NEXT loops in program listings.

EXAMPLE

INDENT = 5

```
10  FOR X = 1 TO 100
20  PRINT X
30  NEXT X
```

LIST

In the above example, the reserved variable **INDENT** is assigned a value of 5 in direct mode before entering the program.

RESULT

The LIST command in direct mode returns the following indented display:

```
10  FOR X = 1 TO 100
20       PRINT X
30       NEXT X
```

5 spaces have been used to indent the above loop.

NOTES

- The system's default value is set to 2 spaces.

INPUT

TYPE Statement

FORMAT **INPUT** [*string*, |;] *variable* {, *variable*}

ACTION Prints the prompt string (if present) on the screen; halts program execution and waits for input from the keyboard; assigns each item as it is input to the next variable in the variable list.

variable may be the name of a numeric, a string, or an array variable. Data items entered from the keyboard must be of the same type (numeric/string) as the corresponding variables. They must be separated by commas, and their number must be the same as the number of *variables* in the list.

EXAMPLE

```
10 REM *** INPUT SUBROUTINE
20 INPUT "Customer order number:"; ORDER%
30 INPUT "Item number:"; ITEM%
40 INPUT "Quantity:"; QUANT%
50 RETURN
```

RESULT

Line 10: Remarks to document subroutine.

20: Prints the string on the screen and assigns input to ORDER%.

30: Prints the string on the screen and assigns input to ITEM%.

40: Prints the string on the screen and assigns input to QUANT%.

50: Returns execution to the main program.

NOTES

- Multiple data items typed on the same input line must be separated by commas.
- Pressing the RETURN or ENTER key signals the end of the input line.
- A question mark is usually printed to prompt the user. You may use a comma instead of a semicolon after the prompt to suppress the question mark.

INPUT#

TYPE File statement

FORMAT **INPUT#** *file number* [, *record number*]; *variable* [{, *variable*}]

ACTION Reads a TEXT file whose reference number is specified following the number sign.

file number is the number used when the file was opened for input.

EXAMPLE

INPUT#1,32;A,B%,C$

COMMENTS

- *record number* following the file reference number specifies where reading should start.
- A comma separates the file number from the record number.
- A semicolon must separate the record number from the variable list.
- **INPUT#** reads a line of text for each variable in its list of variables. A comma must separate each variable.
- The variable list above consists of a real variable (A), an integer variable (B%), and a string variable (C$).

NOTES

- **INPUT#** automatically performs any necessary string to numeric-type conversions, similar to the VAL function.
- You may open a directory as you would a TEXT file by specifying the pathname. Like a CATALOG statement, **INPUT#** may then access the directory to return its data one line at a time.

INSTR

stands for **IN** *and* **STRING**

TYPE String function

FORMAT **INSTR** (*subject string, target string* [, *starting position*])

ACTION Returns a number representing the position of *target string* within *subject string*.

The optional *starting position* is a numeric expression; *subject string* and *target string* are string expressions. The value returned by the **INSTR** function is the numeric value of the position of target string's first character within *subject string*. Searching is done from *starting position*. If *target string* is not found, the returned numeric value is 0 (zero).

EXAMPLE

1. *string expression* can be a string constant;

PRINT **INSTR** ("BLABLABLA", "A", 1)	Returns 3
PRINT **INSTR** ("BLABLABLA", "A", 4)	Returns 6
PRINT **INSTR** ("BLABLABLA", "A", 7)	Returns 9

2. a string variable.

A$ = "BLABLABLA" : B$ = "A"

PRINT **INSTR** (A$, B$, 1)	Returns 3
PRINT **INSTR** (A$, B$, 4)	Returns 6
PRINT **INSTR** (A$, B$, 7)	Returns 9

NOTES

- *starting position* should not be larger than the maximum string length, which is 255 characters.
- Business BASIC has 12 string or string-related functions: ASC, CHR$, HEX$, **INSTR**, LEFT$, LEN, MID$, RIGHT$, STR$, SUB$, TEN, VAL.

INT

stands for **INTEGER**

TYPE Numeric function

FORMAT **INT** (*arithmetic expression*)

ACTION Returns the largest integer smaller than or equal to *arithmetic expression*.

Numeric functions may be used either in immediate mode in conjunction with a PRINT statement or in deferred execution. The argument to all numeric functions must be an *arithmetic expression.* All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions.

EXAMPLE

1. *arithmetic expression* can be a numeric constant;

   PRINT **INT** (1.234) Returns 1
   PRINT **INT** (12.345) Returns 12

2. a numeric variable;

   A = 25.65 : B = −25.65
   PRINT **INT** (A) Returns 25
   PRINT **INT** (B) Returns −26

3. an arithmetic operation;

   PRINT 10 + **INT** (20.36) Returns 30
   PRINT **INT** (12.1 * 6) Returns 72

4. any valid combination thereof.

   A = −25.65 : B = 30.36−20
   PRINT 10 + **INT** (A + B + (10.1 * 6)) Returns 55

NOTES

- Business BASIC has 16 numeric functions in the following type categories:

trigonometric:	ATN, COS, SIN, TAN
arithmetic:	ABS, EXP, **INT**, LOG, RND, SGN, SQR
conversion:	CONV, CONV%, CONV&, CONV$
user-defined:	DEF FN

INVERSE

TYPE Statement

FORMAT **INVERSE**

ACTION Sets screen output in the inverse mode.

EXAMPLE

```
10 HOME
20 INVERSE
30 PRINT "BLACK characters on a WHITE background"
40 FOR T = 1 TO 1000 : NEXT T
50 NORMAL
60 PRINT "WHITE characters on a BLACK background"
70 END
```

RESULT

Line 10: Clears the screen and sets the cursor to the upper-leftmost position.

20: Sets the inverse display mode.

30: Displays the string:

BLACK characters on a WHITE background

40: Delay loop.

50: Restores the normal display mode.

60: Displays the string:

WHITE characters on a BLACK background

NOTES

- No parameter is required after **INVERSE**.
- **INVERSE** may be used either in the immediate (command) mode by typing **INVERSE** and pressing the RETURN or ENTER key or in the deferred (program) mode with a line number.

KBD *stands for* **KEYBOARD**

TYPE Reserved variable

FORMAT **KBD**

ACTION Contains the ASCII code number of the last key pressed on the keyboard.

EXAMPLE

```
 10  ON KBD GOTO 100
 20  GOTO 10
100  PRINT KBD
110  IF KBD = 65 THEN END
190  ON KBD GOTO 100
200  RETURN
```

RESULT

Line 10: Program execution is transferred to line 100 when any key is pressed.

100: PRINT returns the ASCII code number of the key.

110: If the key struck is a capital A (ASCII code number = 65), the END statement is executed and the program halts.

190: The ON **KBD** statement is re-enabled.

200: The RETURN statement branches program execution to the statement following ON **KBD**, that is, line 20.

20: Unconditional transfer to line 10.

NOTES

- The last statement of a subroutine to which program execution has been transferred with ON **KBD** must always be a RETURN statement.
- The ON **KBD** statement must be re-enabled (executed) just before the RETURN statement.

LEFT$

TYPE String function

FORMAT **LEFT$** (*string expression*, *number of characters*)

ACTION Returns the leftmost *number of characters* of *string expression*.

EXAMPLE

1. *string expression* can be a string constant;

PRINT **LEFT$** ("AFTERNOON",3)	Returns AFT
PRINT **LEFT$** ("AFTERNOON",5)	Returns AFTER

2. a string variable;

A$ = "AFTERNOON"	
PRINT **LEFT$** (A$,3)	Returns AFT
PRINT **LEFT$** (A$,5)	Returns AFTER

3. any valid combination thereof.

A$ = "AFTER" : A = 5	
PRINT **LEFT$** (A$ + "NOON",A)	Returns AFTER

NOTES

- If *number of characters* is greater than the total length of *string expression*, the entire string is returned. If *number of characters* = 0, the null string ("") is returned.
- The maximum string length is 255 characters.
- Business BASIC has 12 string or string-related functions: ASC, CHR$, HEX$, INSTR, **LEFT$**, LEN, MID$, RIGHT$, STR$, SUB$, TEN, VAL.

LEN

stands for **LENGTH**

TYPE String function

FORMAT **LEN** (*string expression*)

ACTION Returns the length (number of characters) of *string expression*.

EXAMPLE

1. *string expression* can be a string constant;

   PRINT **LEN** ("AFTERNOON") — Returns 9
   PRINT **LEN** ("A1F2T + E/R%N!O O N") — Returns 17

2. a string variable;

   A$ = "AFTER" : B$ = "NOON"
   PRINT **LEN** (A$) — Returns 5
   PRINT **LEN** (B$) — Returns 4

3. any valid combination thereof.

   A$ = "AFTER" : B$ = "NOON"
   PRINT **LEN** (A$ + B$) — Returns 9

NOTES

- The **LEN** function returns an integer number.
- **LEN** counts all characters including blank spaces.
- The number of characters in a string expression may range from 0 (zero) to 255. A null string is a string that contains no characters.
- A string variable is identified by a dollar sign ($).
- Business BASIC has 12 string or string-related functions: ASC, CHR$, HEX$, INSTR, LEFT$, **LEN**, MID$, RIGHT$, STR$, SUB$, TEN, VAL.

LESS THAN

symbol <

TYPE Relational operator

FORMAT *expression1* < *expression2*

ACTION Allows a logical comparison to be made between two expressions.

expression1 and *expression2* are either both numeric or both string. The comparison returns a logical value. If the value of *expression1* is less than the value of *expression2*, the result of the comparison is true (non-zero, represented by the numerical value −1); otherwise, the result is false (represented by 0). Relational operators are usually used in conditional statements, such as IF ... THEN statements, to make a decision regarding program flow.

EXAMPLE

```
10 A = 10 : B = 20 : C = 2 : X$ = "TRUSTY" : Y$ = "TRUST"
20 IF A < B THEN PRINT "TRUE" : ELSE PRINT "FALSE"
30 IF A < B/C THEN PRINT "TRUE" : ELSE PRINT "FALSE"
40 IF X$ < Y$ THEN PRINT "TRUE" : ELSE PRINT "FALSE"
```

RESULT

Line 10: Assigns values to the numeric variables A, B, C, and the string variables X$ and Y$.

20: Since A is less than B, prints: TRUE.

30: Since A is not less than B divided by C, prints: FALSE.

40: Since TRUSTY is not smaller than TRUST, prints: FALSE.

NOTES

- The strings are compared character by character, from left to right, on the basis of their ASCII code numbers. The first character found in one string that has a greater ASCII value than the character found in the same position in the second string makes the first string greater. If the characters in the same positions are identical but one string's current length is longer, the longer string is greater.
- Business BASIC has 6 relational operators:

=	Equal to
<> *or* ><	Not equal to
>	Greater than
>= *or* =>	Greater than or equal to
<	Less than
<= *or* =<	Less than or equal to

LESS THAN OR EQUAL TO *symbol* <= *or* =<

TYPE Relational operator

FORMAT *expression1* <= *expression2*

ACTION Allows a logical comparison to be made between two expressions.

expression1 and *expression2* are either both numeric or both string. The comparison returns a logical value. If the value of *expression1* is less than or equal to the value of *expression2*, the result of the comparison is true (non-zero, represented by the numerical value −1); otherwise, the result is false (represented by 0). Relational operators are usually used in conditional statements, such as IF ... THEN statements, to make a decision regarding program flow.

EXAMPLE

```
10  A = 10 : B = 20 : C = 2 : X$ = "TRUSTY" : Y$ = "TRUST"
20  IF A <= B THEN PRINT "TRUE" : ELSE PRINT "FALSE"
30  IF A <= B/C THEN PRINT "TRUE" : ELSE PRINT "FALSE"
40  IF X$ <= Y$ THEN PRINT "TRUE" : ELSE PRINT "FALSE"
```

RESULT

Line 10: Assigns values to the numeric variables A, B, C, and the string variables X$ and Y$.

20: Since A is less than B, prints: TRUE.

30: Since A is equal to B divided by C, prints: TRUE.

40: Since TRUSTY is not smaller or equal to TRUST, prints: FALSE.

NOTES

- The strings are compared character by character, from left to right, on the basis of their ASCII code numbers. The first character found in one string that has a greater ASCII value than the character found in the same position in the second string makes the first string greater. If the characters in the same positions are identical but one string's current length is longer, the longer string is greater.
- Business BASIC has 6 relational operators:

=	Equal to
<> *or* ><	Not equal to
>	Greater than
>= *or* =>	Greater than or equal to
<	Less than
<= *or* =<	Less than or equal to

LET

TYPE Assignment statement

FORMAT [**LET**] *variable* | *reserved variable* = *value*

ACTION Assigns *value* to the variable specified by *variable name*.

The type of *value* (string or numeric) must match the type of *variable*.

EXAMPLE

```
10 LET A = 10
20 LET B = A + 10
30 LET C = (A * B)/2
40 LET L$ = "THE BASIC LANGUAGE"
50 PRINT A, B, C, L$
60 END
```

RESULT

Line 10: Variable A is assigned the value 10.

20: Variable B is assigned the result of the addition.

30: Variable C is assigned the result of the mathematical operation.

40: Variable L$ is assigned the string THE BASIC LANGUAGE.

50: The four variables' values are printed out.

NOTES

- The keyword **LET** is optional.

    variable name = *value*

  and

    LET *variable name* = *value*

  are equivalent statements.
- Although *variable name* = *value* looks like a relational expression, it is interpreted by BASIC as an assignment statement, and has no logical value.
- Programmers sometimes use **LET** to emphasize lines where a new value is assigned to a variable.

LIST

TYPE Statement

FORMAT **LIST** [*line number1*] [TO | , | –[*line number2*]]

ACTION Lists one or more program lines on the screen or other specified device.

line number must be in the range from 0 (zero) to 65529. *line number1* is the first line to be listed. *line number2* is the last line to be listed.

EXAMPLE

In the immediate mode:

1. **LIST** 10
2. **LIST** 10–50
3. **LIST** –50
4. **LIST** 50–

In the deferred (program) mode:

10: INPUT X : IF X = 1 THEN **LIST** 10–100

RESULT

In the immediate mode:

1. Lists line 10.
2. Lists all lines numbered from 10 to 50 inclusive.
3. Lists all lines from the beginning of the program until line 50 inclusive.
4. Lists all lines from line 50 to the end of the program.

In the deferred (program) mode:

Lists all lines from line 10 to 100 inclusive, if the INPUT value at line 10 is equal to 1.

NOTES

- The listing (display) can be temporarily halted in the immediate mode by pressing the CONTROL key followed by the letter C.

LOAD

TYPE File statement

FORMAT **LOAD** *pathname*

ACTION Reads a specified BASIC program from a disk file and stores it in memory.

EXAMPLE

1. **LOAD** .D1/Inventory
2. **LOAD**/Accounting/Inventory

COMMENTS

- The disk drive reference name consists of a period, the letter D, and the drive number. .D1 refers to the built-in disk drive. .D2, .D3, and .D4 will refer to additional external disk drives.
- The volume name or the file name must be preceded by a slash (/).

NOTES

- When a **LOAD** command is executed, the numeric variables are automatically set to 0 (zero) and the string variables to null strings. All files are closed with the exception of any EXEC file being executed. Any program currently stored in memory is erased and replaced by the new program.
- If an error is made, the following messages are displayed: ?UNDEF'D STATEMENT ERROR, if the specified line number does not exist; ?TYPE MISMATCH ERROR, if the specified file is not a BASIC program; ?FILE NOT FOUND ERROR, if the specified file name does not exist.

LOCK

TYPE File statement

FORMAT **LOCK** *pathname*

ACTION Protects a file from being inadvertently deleted, changed, or renamed.

The **LOCK** statement must be followed by the file or subdirectory name you wish to lock.

EXAMPLE

LOCK/Purchases/Suppliers/France

NOTES

- When listed by a CATALOG statement, locked files are shown with an asterisk (*) to the left of their file type.

Type	*Blks*	*Name*
*BASIC	00003	TRANSACTIONS
*DATA	00015	PHONE.NUMBERS
*FOTO	00009	STATISTICS

- To protect all the files on a disk, a tab may be placed over the write-protect cutout on the upper-right edge of the disk.

LOG

stands for **LOGARITHM**

TYPE Numeric function

FORMAT **LOG** (*arithmetic expression*)

ACTION Returns the natural logarithm of *arithmetic expression*.

Numeric functions may be used either in immediate mode in conjunction with a PRINT statement or in deferred execution. The argument to all numeric functions must be an *arithmetic expression.* All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions.

EXAMPLE

1. *numeric expression* can be a numeric constant;

PRINT **LOG** (2)	Returns .69314718
PRINT **LOG** (6 − 2)	Returns 1.38629436

2. a numeric variable;

A = 2 : B = 6	
PRINT **LOG** (A)	Returns .69314718
PRINT **LOG** (B − A)	Returns 1.38629436

3. any valid combination thereof.

FOR J = 2 TO 6 STEP 2	Returns .69314718
PRINT **LOG** (J)	Returns 1.38629436
NEXT J	Returns 1.79175947

NOTES

- *arithmetic expression* must be greater than 0 (zero): **LOG** (0) or **LOG** (−2) returns an "Illegal Quantity" error message. The natural logarithm is the logarithm to the base *e*.
- Business BASIC has 16 numeric functions in the following type categories:

trigonometric:	ATN, COS, SIN, TAN
arithmetic:	ABS, EXP, INT, **LOG**, SGN, SQR, RND
conversion:	CONV, CONV%, CONV&, CONV$
user-defined:	DEF FN

MID$ *stands for* **MIDDLE**

TYPE String function

FORMAT **MID$** (*string, starting position* [, *number of characters*])

ACTION Returns the requested number of characters of a string expression, starting at a specified character position.

string is a string expression; **MID$** is used to extract a section of *string*. *starting position* is a numeric expression specifying the first (leftmost) character in the substring; *number of characters* is a numeric expression specifying the length of the substring.

EXAMPLE

1. *string expression* can be a string constant;

PRINT **MID$** ("AFTERNOON",6,4)	Returns NOON
PRINT **MID$** ("AFTERNOON",1,5)	Returns AFTER

2. a string variable;

A$ = "AFTERNOON"	
PRINT **MID$** (A$,6,4)	Returns NOON
PRINT **MID$** (A$,1,5)	Returns AFTER

3. any valid combination thereof.

A$ = "AFTER" : A = 6	
PRINT **MID$** (A$ + "NOON",A,4)	Returns NOON

NOTES

- The number of characters in a string expression may range from 0 (zero) to 255.
- A null string is a string that contains no characters.
- A string variable is identified by a dollar sign ($).
- Business BASIC has 12 string or string-related functions: ASC, CHR$, HEX$, INSTR, LEFT$, LEN, **MID$**, RIGHT$, STR$, SUB$, TEN, VAL.

MOD

stands for **MODULO**

TYPE Arithmetic operator

FORMAT *numeric expression1* **MOD** *numeric expression2*

ACTION Returns the integer value that is the remainder of the integer division of *numeric expression1* by *numeric expression2*.

EXAMPLE

1. *numeric expression* can be a numeric constant;

PRINT 4 **MOD** 3	Returns 1
PRINT 27 **MOD** 4	Returns 3
PRINT 45 **MOD** 8	Returns 5

2. a numeric variable;

A = 4 : B = 3 : C = 27 : E = 8

PRINT A **MOD** B	Returns 1
PRINT C **MOD** A	Returns 3
PRINT 45 **MOD** E	Returns 5

3. or an arithmetic operation.

A = 27 : B = 2

PRINT A **MOD** (B * B)	Returns 3

NOTES

- Business BASIC has 9 arithmetic operators:

+	Unary plus
–	Unary minus
^	Exponentiation
*	Multiplication
/	Floating-point division
MOD	Modulo division
DIV	Integer division
+	Addition
–	Subtraction

MULTIPLICATION *symbol* *

TYPE Arithmetic operator

FORMAT *numeric expression1* * *numeric expression2*

ACTION Performs an arithmetic multiplication.

EXAMPLE

1. *numeric expression* can be a numeric constant;

PRINT 20 * 10	Returns 200
PRINT −20 * (−10)	Returns 200

2. a numeric variable;

A = 20 : B = 10	
PRINT A * B	Returns 200
PRINT −A * (−B)	Returns 200

3. any valid combination thereof.

A = 20 : B = 10	
PRINT A * 10	Returns 200
PRINT −20 * (−B)	Returns 200

NOTES

- Business BASIC has 9 arithmetic operators:

+	Unary plus
−	Unary minus
^	Exponentiation
*	Multiplication
/	Floating-point division
MOD	Modulo division
DIV	Integer division
+	Addition
−	Subtraction

NEW

TYPE Statement

FORMAT **NEW**

ACTION Erases the program currently stored in memory, clears all variables, and closes all open files.

EXAMPLE

```
  10  GOSUB 1000
  20  END
       |
1000  REM *** Subroutine to enter new program
1010  INPUT "Do you want to erase the current program"; X$
1020  IF X$ = "YES" THEN NEW : ELSE RETURN
```

RESULT

Line 10: Unconditional transfer to line 1000.

20: Ends the program.

1000: Remarks to document program.

1010: Asks a question, accepts the input, and assigns it to the variable X$.

1020: If the answer at 1010 is YES, the program will be erased; otherwise, program execution will return to the next executable statement following the last executed GOSUB.

NOTES

- **NEW** may be used in the immediate (command) mode by typing **NEW** and pressing the RETURN or ENTER key.
- When a program is loaded from a peripheral unit, the program stored in the computer's memory is erased and replaced by the new one.

NEXT

TYPE Statement

FORMAT FOR *control variable* = *aexpr1* TO *aexpr2* [STEP *aexpr3*]

|

NEXT [*control variable*{, *control variable*}]

ACTION Sets up a program loop that repeats the series of instructions inside the loop a given number of times.

aexpr is an *arithmetic expression.* The loop begins with the FOR statement and ends with the **NEXT** statement. Every statement in between is executed once with each repetition. Every repetition automatically increments (adds to) the value of *control variable* by a value equal to *aexpr3*; if STEP is omitted, the default increment is 1. *control variable* starts off having a value equal to *aexpr1*; when the value of *control variable* reaches *aexpr2*, the loop is ended and program execution continues with the statement after **NEXT**. A conditional statement can be used to exit the loop before it is finished.

EXAMPLE

```
10 FOR B = 1 TO 10
20 PRINT "AZ";
30 NEXT B
40 END
```

RESULT

Line 10: Sets up a loop to repeat 10 times.

20: Prints string AZ.

30: Repeats from line 10.

NOTES

- The initial value of *control variable* B has been incremented by the default value of 1.
- A loop structure may contain other loops within it, provided that the loops are nested.

NORMAL

TYPE Statement

FORMAT **NORMAL**

ACTION Resets the screen output in the normal mode.

EXAMPLE

```
10 HOME
20 INVERSE
30 PRINT "BLACK characters on a WHITE background"
40 FOR T = 1 TO 1000 : NEXT T
50 NORMAL
60 PRINT "WHITE characters on a BLACK background"
70 END
```

RESULT

Line 10: Clears the screen and sets the cursor to the upper-leftmost position.

20: Sets the inverse display mode.

30: Displays the string:

BLACK characters on a WHITE background

40: Delay loop.

50: Restores the normal display mode.

60: Displays the string:

WHITE characters on a BLACK background

NOTES

- No parameter is required after **NORMAL**.
- **NORMAL** may be used either in the immediate (command) mode by typing **NORMAL** and pressing the RETURN or ENTER key, or in the deferred (program) mode with a line number.

NOT

TYPE Logical operator

FORMAT **NOT** (*expression*)

ACTION Reverses the logical evaluation of an expression. The relational value of a comparison between two expressions (numeric or string) is represented by the numerical value of −1 if the relationship is true, and 0 (zero) if the relationship is false.

EXAMPLE

```
10  PRINT (100 < 50)
20  PRINT NOT (100 < 50)
30  END
```

RESULT

Line 10: The logical expression (100 is less than 50) is evaluated to false; prints 0 (zero).

20: The logical expression (100 is less than 50) is evaluated to true since **NOT** has reversed its logical evaluation; prints −1.

NOTES

- The strings are compared character by character, from left to right, on the basis of their ASCII code numbers. The first character found in one string that has a greater ASCII value than the character found in the same position in the second string makes the first string greater. If the characters in the same positions are identical but one string's current length is longer, the longer string is greater.
- Business BASIC has three logical operators:

AND	Conjunction
OR	Inclusive disjunction
NOT	Negation (logical complement)

NOT EQUAL TO *symbol* <> *or* ><

TYPE Relational operator

FORMAT *expression1* <> *expression2*

ACTION Allows a logical comparison to be made between two expressions.

expression1 and *expression2* are either both numeric or both string. The comparison returns a logical value. If *expression1* does not have the same value as *expression2*, the result of the comparison is true (non-zero, represented by the numerical value −1); otherwise, the result is false (represented by the numerical value 0). Relational operators are usually used in conditional statements, such as IF ... THEN statements, to make a decision regarding program flow.

EXAMPLE

```
10  A = 10 : B = 20 : C = 2 : X$ = "TRUSTY" : Y$ = "TRUST"
20  IF A <> B THEN PRINT "TRUE" : ELSE PRINT "FALSE"
30  IF A <> B/C THEN PRINT "TRUE" : ELSE PRINT "FALSE"
40  IF X$ <> THEN PRINT "TRUE" : ELSE PRINT "FALSE"
```

RESULT

Line 10: Assigns values to the numeric variables A, B, C, and the string variables X$ and Y$.

20: Since A is not equal to B, prints: TRUE.

30: Since A is equal to B divided by C, prints: FALSE.

40: Since TRUSTY is not equal to TRUST, prints: TRUE.

NOTES

- The strings are compared character by character, from left to right, on the basis of their ASCII code numbers. The first character found in one string that has greater ASCII value than the character found in the same position in the second string makes the first string greater. If the characters in the same positions are identical but one string's current length is longer, the longer string is greater.
- Business BASIC has 6 relational operators:

=	Equal to
<> *or* ><	Not equal to
>	Greater than
>= *or* =>	Greater than or equal to
<	Less than
<= *or* =<	Less than or equal to

NOTRACE

TYPE Statement

FORMAT **NOTRACE**

ACTION Cancels the TRACE statement.

TRACE is used mainly to debug (check, troubleshoot) the sequential execution of a program or parts of it. During program execution, TRACE displays a number sign (#) followed by the line numbers of the statements in the sequential order of their execution. After **NOTRACE** is executed, the line numbers of executing program statements are not displayed.

EXAMPLE

1.
```
10 A = 25
20 B = 55
30 C = A + B
40 PRINT C

TRACE

RUN
```

2.
```
NOTRACE

RUN
```

RESULT

1. The TRACE command will cause the following display:

   #10 #20 #30 #40 80

2. The **NOTRACE** command will cancel the traced execution. Only the result of the PRINT statement is displayed:

   80

NOTES

- **NOTRACE** may be used either in the immediate (command) mode or in the deferred mode.
- Traced execution of assignment statements is denoted only by the statements' line numbers. If the traced statement contains a PRINT statement, TRACE displays the line number and the result of the PRINT statement.

OFF EOF# *stands for* OFF *and* END OF FILE

TYPE File statement

FORMAT **OFF EOF#** *file number*

ACTION Cancels an ON EOF# statement.

The ON EOF# statement allows program execution to branch to a statement or statement list when execution continues past the end of a specified file. After an **OFF EOF#** statement has been executed, Business BASIC resumes displaying error messages and halting execution when an end of file is reached, just as it did before the ON EOF# statement was executed.

EXAMPLE

```
10  REM *** ON EOF# statement : File Copy Utility Program
20  INPUT "Type the source file name to be copied "; L$
30 OPEN#1 AS INPUT, L$
40 INPUT "Type the copy file name to print to"; L$
50 OPEN#2 AS OUTPUT, L$
60 ON EOF#1 PRINT "Copy completed"
70 CLOSE
80 END
90 INPUT#1; L$ : PRINT#2; L$ : GOTO 90
```

OFF EOF#1

NOTES

- ON EOF# is very similar to the ON ERR statement, except that ON EOF# recognizes only one error code. Unlike ON ERR, you cannot use the RESUME statement with ON EOF# statements.
- If a program reads past the end of a file and ON EOF# is not in effect, program execution halts and the ?OUT OF DATA ERROR message is displayed.

OFF ERR

stands for **OFF** *and* **ERROR**

TYPE Statement

FORMAT **OFF ERR**

ACTION Cancels the most recently executed ON ERR statement.

ON ERR causes program execution to branch to a specified line number. If an error occurs after an **OFF ERR** statement, program execution stops and an error message is displayed.

EXAMPLE

```
10 ON ERR GOTO 70
20 DIM A (12)
30 FOR X = 1 TO 12 : READ A : NEXT X
40 GOTO 80
50 DATA 1, 2, 3, 4, 5, 6
60 END
70 IF ERR = 4 THEN RESUME 40
80 PRINT "Program execution continues"
```

OFF ERR

NOTES

- **OFF ERR** may be used either in the immediate (command) mode or in the deferred mode.
- **OFF ERR** has no parameters or options.

OFF KBD

stands for **OFF** *and* **KEYBOARD**

TYPE Statement

FORMAT **OFF KBD**

ACTION Cancels the ON KBD statement.

ON KBD causes program execution to branch to the line number specified after the GOTO or GOSUB statements when any key is pressed.

EXAMPLE

```
 10 ON KBD GOTO 100
 20 GOTO 10
100 PRINT KBD
110 IF KBD = 65 THEN END
190 ON KBD GOTO 100
200 RETURN
```

OFF KBD

NOTES

- **OFF KBD** may be used in the immediate (command) mode or in the deferred mode.
- CONTROL-C cannot halt program execution when the ON KBD statement is in effect. CONTROL-C is treated just like any other key.

ON EOF# *stands for* **ON** *and* **END OF FILE**

TYPE File statement

FORMAT **ON EOF#** *file number* | *statement list*

ACTION Allows program execution to branch to a statement or statement list when execution continues past the end of a specified file.

EXAMPLE

```
10 REM *** ON OEF# statement : File Copy Utility Program
20 INPUT "Type the source file name to be copied "; L$
30 OPEN#1 AS INPUT, L$
40 INPUT "Type the copy file name to print to "; L$
50 OPEN#2 AS OUTPUT, L$
60 ON EOF#1 PRINT "Copy completed"
70 CLOSE
80 END
90 INPUT#1; L$ : PRINT#2; L$ : GOTO 90
```

RESULT

Line 10: Documents program.

20: Prints the message and assigns the source file name to the string variable L$.

30: Opens the source file L$ as a read-only file whose reference number is #1.

40: Prints the message and assigns the copy file name to the string variable L$.

50: Opens the copy file L$ as a write-only file whose reference number is #2.

60: Displays the string: "Copy completed" at the **EOF** of file #1.

70: Closes both files (#1 and #2).

90: Sets up a copying "loop". INPUT#1; L$ reads one line at a time from source file #1 and assigns it to L$. PRINT#2; L$ prints line L$ to copy file #2. GOTO 90 branches back to the beginning of line 90 until the end of file #1 is reached.

NOTES

- **ON EOF#** is very similar to the ON ERR statement, except that **ON EOF#** recognizes only one error code. Unlike ON ERR, you cannot use the RESUME statement with **ON EOF#** statements.
- If a program reads past the end of a file and **ON EOF#** is not in effect, program execution halts and the ?OUT OF DATA ERROR message is displayed.

ON ERR

stands for **ON** *and* **ERROR**

TYPE Statement

FORMAT **ON ERR** *statement*

ACTION Causes program execution to branch to the specified line number.

EXAMPLE

```
10 ON ERR GOTO 70
20 DIM A (12)
30 FOR X = 1 TO 12 : READ A : NEXT X
40 GOTO 80
50 DATA 1, 2, 3, 4, 5, 6
60 END
70 IF ERR = 4 THEN RESUME 40
80 PRINT "Program execution continues"
```

RESULT

Line 10: If an error occurs, **ON ERR** causes an unconditional branching to line 70.

20: Dimensions a 12-element list.

50: Since the DATA statement contains only 6 data items, the unconditional branching **ON ERR** GOTO 70 is executed.

70: Program execution resumes at line 40 (the code number of the ?OUT OF DATA ERROR is 4).

40: GOTO causes an unconditional branching to line 80.

80: Program execution continues at line 80.

NOTES

- **ON ERR** is used only as a program statement. The **ON ERR** statement should be placed at the beginning of a program.
- The error-handling subroutine statements must be free of errors, or an endless and unstoppable loop may result. Error-handling subroutines usually end with a RESUME statement.
- If a program contains more than one **ON ERR** statement, only the most recently executed **ON ERR** statement will be used.

ON ... GOSUB *stands for* **ON, GO** *and* **SUBROUTINE**

TYPE Statement

FORMAT **ON** *arithmetic expression* **GOSUB** *line number* {[, *line number*]}

ACTION Transfers program execution to one of several specified *line numbers* depending on the value of *arithmetic expression*.

ON ... GOSUB allows the program to choose one of several paths; this is called "multiple branching." If the value of *arithmetic expression* is 1, the program jumps to the first *line number* in the list; if the value is 2, the program jumps to the second *line number* in the list, and so on. When a RETURN statement is next encountered (in the subroutine to which program execution jumps), program execution will return to the next executable statement after **ON ... GOSUB**.

EXAMPLE

```
 10  INPUT X
 20  ON X GOSUB 100, 200, 300 : END
     |
100  PRINT "First line number in the list" : RETURN
200  PRINT "Second line number in the list" : RETURN
300  PRINT "Third line number in the list" : RETURN
```

RESULT

Line 10: Asks for input and assigns it to variable X.

20: Sends program execution down on one of three branches: if X is 1, jumps to line 100; if X is 2, jumps to line 200; if X is 3, jumps to line 300.

NOTES

- If the value of *arithmetic expression* is 0 (zero) or greater than 3, program execution branches to the first *line number* in the list.
- *arithmetic expression*, which is rounded to an integer, must be in the range from 0 (zero) to 255.

ON ... GOTO

TYPE Statement

FORMAT **ON** *arithmetic expression* **GOTO** *line number* {[, *line number*]}

ACTION Transfers program execution to one of several specified *line numbers* depending on the value of *arithmetic expression*.

ON ... GOTO allows the program to choose one of several paths; this is called "multiple branching." If the value of *arithmetic expression* is 1, the program jumps to the first *line number* in the list; if the value is 2, the program jumps to the second *line number* in the list, and so on.

EXAMPLE

```
 10  INPUT X
 20  ON X GOTO 100, 200, 300
      |
100  PRINT "First line number in the list" : END
200  PRINT "Second line number in the list" : END
300  PRINT "Third line number in the list" : END
```

RESULT

Line 10: Asks for input and assigns it to variable X.

20: Sends program execution down on one of three branches: if X is 1, jumps to line 100; if X is 2, jumps to line 200; if X is 3, jumps to line 300.

NOTES

- If the value of *arithmetic expression* is 0 (zero) or greater than 3, program execution branches to the first *line number* in the list.
- Each *line number* in the list following the **ON ... GOTO** statement must be the first *line number* of the module you wish to branch to.

ON KBD

stands for **ON** *and* **KEYBOARD**

TYPE Statement

FORMAT **ON KBD** *statement*

ACTION Causes program execution to branch to the line number specified after the GOTO or GOSUB statements when any key is pressed.

EXAMPLE

```
 10 ON KBD GOTO 100
 20 GOTO 10
100 PRINT KBD
110 IF KBD = 65 THEN END
190 ON KBD GOTO 100
200 RETURN
```

RESULT

Line 10: Program execution is transferred to line 100 when any key is pressed.

100: Returns the ASCII code number of the key.

110: If the key struck is capital A (ASCII code number = 65), the END statement is executed and program halts.

190: The **ON KBD** statement is re-enabled.

200: The RETURN statement branches program execution to the statement following **ON KBD**, that is, line 20.

20: Unconditional transfer to line 10.

NOTES

- The last statement of a subroutine to which program execution has been transferred with **ON KBD** must always be a RETURN statement.
- The **ON KBD** statement must be re-enabled (executed) just before the RETURN statement.
- CONTROL-C cannot halt program execution when the **ON KBD** statement is in effect. CONTROL-C is treated just like any other key.

OPEN#

TYPE File statement

FORMAT **OPEN#** *file number* [AS INPUT|AS OUTPUT|AS EXTENSION], *pathname* [, *record size*]

ACTION Opens files for access.

Before a file can be accessed (used), it must be opened with an **OPEN#** statement. All Input/Output statements referring to a file while it is open must specify the same file reference number that has been used to open the file by the **OPEN#** statement.

EXAMPLE

1. OPEN/Customers
2. 100 **OPEN#**1, "Customers"
   200 **OPEN#**3, "Accounts"
3. 100 **OPEN#**1 AS EXTENSION, "Customers"
   200 **OPEN#**7 AS INPUT, ".Console"
   300 **OPEN#**9 AS OUTPUT, ".Printer"

COMMENTS

- In immediate mode, *pathname* need not be enclosed in quotation marks.
- **OPEN#** must be followed by *file number* and *pathname*, separated by a comma. *pathname* must be enclosed in quotation marks.
- The reserved words AS INPUT and AS OUTPUT specify that the file is opened as a read-only or write-only file, respectively.

  The AS EXTENSION option is used in sequential access to append new information to an existing file.

  A period must precede device names.

NOTES

- *file number* may be any arithmetic expression from 1 to 10.
- Only up to 10 files may be opened at the same time.
- If an **OPEN#** statement contains a file reference number equal to one presently in use, the first file using that file reference number is automatically closed.

OR

TYPE Logical operator

FORMAT *condition1* **OR** *condition2*

ACTION Connects two or more conditions.

The expression evaluates as true (non-zero) if one of the conditions is true; otherwise, it evaluates as false (zero). The result of the evaluation is then usually used in conditional statements, such as IF ... THEN statements, to make a decision regarding program flow.

EXAMPLE

```
10 A = 10 : B = 50 : C = 100
20 IF A < B OR C = B THEN 40
30 PRINT "NEITHER OF THE TWO CONDITIONS HAS BEEN
   MET" : END
40 PRINT "ONE OF THE TWO CONDITIONS HAS BEEN MET"
50 A$ = "A" : B$ = "B" : C$ = "B"
60 IF A$ = B$ OR C$ <> B$ THEN 80
70 PRINT "ONE OF THE TWO CONDITIONS HAS BEEN MET" : END
80 PRINT "NEITHER OF THE TWO CONDITIONS HAS BEEN MET"
90 END
```

RESULT

Line 40: One of the two conditions has been met since A is smaller than B; the message on line 40 is printed:

ONE OF THE TWO CONDITIONS HAS BEEN MET

80: Neither of the conditions has been met since A$ = "A" is different from B$ = "B" and B$ = C$; the message on line 80 is printed:

NEITHER OF THE TWO CONDITIONS HAS BEEN MET

NOTES

- The strings are compared character by character, from left to right, on the basis of their ASCII code numbers. The first character found in one string that has a greater ASCII value than the character found in the same position in the second string makes the first string greater. If the characters in the same positions are identical but one string's current length is longer, the longer string is greater.
- Business BASIC has three logical operators:

AND	Conjunction
OR	Inclusive disjunction
NOT	Negation (logical complement)

OUTPUT#

TYPE File statement

FORMAT **OUTPUT#** *file number*

ACTION Directs screen output to a specified file.

All PRINT, LIST, TRACE, and CATALOG statement output is sent to the specified device file.

EXAMPLE

1. **OUTPUT#** 1
2. **OUTPUT#** Ø

RESULT

1. The file reference number following the **OUTPUT#** statement must be identical to the file number specified in the OPEN# statement.
2. **OUTPUT#** Ø causes normal screen output to be resumed. Business BASIC treats as a *file* any peripheral device that is connected to your Apple. Ø is the screen's file reference number.

NOTES

- Error messages displayed with nonvalid **OUTPUT#** statements are: ?FILE NOT OPEN ERROR, if no file is open with the same reference number; ?TYPE MISMATCH ERROR, if the specified file does not accept characters.
- The TRACE statement should not be used to debug programs using the **OUTPUT#** statement, unless you want the TRACE-generated line numbers sent to the file.

OUTREC

stands for **OUT** *and* **RECORD**

TYPE Reserved variable

FORMAT **OUTREC** = *arithmetic expression*

ACTION Contains the maximum length of lines output on a printer by the LIST command.

EXAMPLE

OUTREC = 78

RESULT

Printer starts a new line as soon as the specified column position (78, in the example) assigned to **OUTREC** is reached.

NOTES

- The value of **OUTREC** must be greater than the value of INDENT.

PARENTHESES *symbols* **( )**

TYPE Operator

FORMAT **(***arithmetic expression* {[*arithmetic expression*]}**)**

ACTION Used to define the specific value that is currently being operated on.

A function operates on a value specified by *arithmetic expression*, which is called the "argument" of the function. In expressions made up of multiple operations, the order in which operations are performed can affect the results. There is a standard (default) priority order, but enclosing an operation in parentheses allows you to specify which operations you want performed first.

EXAMPLE

1. PRINT FRE **(**0**)**
2. P = INT **(**X**)**
3. DIM D **(**14,6**)**
4. PRINT TAB **(**10**)**; "ABCD"
5. PRINT SPC **(**Y**)**; "ABCD"
6. C$ = CHR$ **(**65**)**
7. X = **((**2 * 3 + 4 ^ 2**)** * 2 +**)** * **(**32 − 4**)**

RESULT

1–6. The arguments of a function are usually enclosed inside parentheses.

7. The mathematical operations will be performed from left to right in the following order: first, within pairs of parentheses in the order the computer encounters them; and, within the parentheses, in the priority order of the arithmetic operators. The result is 1260.

NOTES

- The order of evaluation of arithmetic operators is:

1.	**( )**	Parentheses	4.	*	Multiplication
2.	+	Unary plus		/	Floating-point division
	−	Unary minus		MOD	Modulo division
3.	^	Exponentiation		DIV	Integer division
			5.	+	Addition
				−	Subtraction

PERCENT *symbol* %

TYPE Identifier

FORMAT *variable name*%

ACTION Identifies a numeric variable as being of the integer type.

Variables have identifiers attached to specify which type of number they represent. A variable without an identifier is automatically of the single-precision type.

EXAMPLE

```
10 A = 4
20 B = 3
30 J% = A/B
40 PRINT "The answer as an integer value is ";J%
```

RESULT

Line 10–20: Assigns values to variables A and B (single-precision type).

30: Sets the integer variable **J%** equal to A divided by B.

40: Prints the message and the value of J%:

The answer as an integer value is 1.

NOTES

- When a higher precision value (such as the result of 4 divided by 3) is assigned to a lower precision variable (such as J%), the number will be rounded before being stored and displayed.
- Business BASIC has three identifiers attached to variable names:

&	For variables of the long integer type
%	For variables of the integer type
$	For variables of the string type

POP

TYPE Statement

FORMAT **POP**

ACTION Erases the return address of the last executed GOSUB statement.

When a GOSUB statement is executed, the line number to which the program will return after the next RETURN statement is saved on a "stack"; since multiple GOSUB statements are possible, a RETURN statement always returns the program to the statement after the last executed GOSUB. **POP** "pops" the last return address off the stack; a subsequent RETURN will return the program to the statement following the next-to-last executed GOSUB.

EXAMPLE

```
 10 GOSUB 100 : REM *** First GOSUB
 20 PRINT "Statements following the first GOSUB"
 30 END
    |
100 GOSUB 120 : REM *** Second GOSUB
110 PRINT "Statements following the second GOSUB"
120 POP
130 RETURN
140 END
```

RESULT

Line 10: Branches to the subroutine at line 100.

100: Branches to the subroutine at line 120.

120: Pops the return address of the last GOSUB statement off the stack.

130: Returns to line 20.

20: Prints the string:

Statements following the first GOSUB

30: Ends the program.

NOTES

- The result given in the example describes the order of execution of the program example.
- **POP** is sometimes used (in command mode) in cases where a subroutine has ended prematurely without executing a RETURN, since the return address will otherwise be left on the top of the stack.

PREFIX$

TYPE File reserved variable

FORMAT **PREFIX$** = "*pathname prefix*"

ACTION Contains a partial pathname.

Using the variable **PREFIX$** allows you to locate a file without the inconvenience of having to specify a complete pathname.

EXAMPLE

1. **PREFIX$** = /Customers
2. **PREFIX$** = ".D2"

COMMENTS

- Prefix set to a volume name.
- Prefix set to a device name.

NOTES

- The device name must be enclosed in quotation marks.
- The contents of the reserved variable **PREFIX$** plus a local name as entered through the keyboard by the user is assumed to be the complete pathname of a file.

PRINT

TYPE Statement

FORMAT ?|**PRINT**|[, |;][*expression*]}[, |;]

ACTION Sends the output of a list of expressions to the screen.

list of expression may consist of numeric and/or string expressions, separated by commas or semicolons.

EXAMPLE

1. N = 100 : **PRINT** N
2. A$ = "ABC" : **PRINT** A$
3. **PRINT** N,N
4. **PRINT** N;N
5. **PRINT** N$,N$
6. **PRINT** N$;N$

RESULT

1. Prints the numeric variable value: 100.
2. Prints the string variable value: ABC.
3. When a comma separates two numeric variables, their values are printed at pretabulated printing zones: 100   100.
4. When a semicolon separates two numeric variables, the two values are printed with only one blank before and after each value: 100 100 .
5. When a comma separates two string variables, their values are printed at pretabulated printing zones: ABC   ABC.
6. A semicolon concatenates two strings: ABCABC.

NOTES

- **PRINT** may be used in the immediate (command) mode by typing **PRINT** and pressing the RETURN or ENTER key.
- Punctuation marks such as semicolons and commas may also be used before and/or after *expression*.

PRINT#

TYPE File statement

FORMAT ?# | **PRINT#** *file number* [, *record number*]
[; *expression* [{; *expression*}][;]]

ACTION Writes data sequentially to files.

PRINT# writes a line of text for each expression in its list of expressions.

list of expressions may be numeric and/or string expressions.

EXAMPLE

PRINT#1, 32;C$(1,1),LEFT$(C$(1,1)),A&,A&/12,B%

COMMENTS

- *file number* is specified following the number sign.
- *record number* following *file number* specifies where writing should start.
- A comma separates *file number* from *record number*.
- A semicolon must separate *record number* from *list of expressions*.
- A comma must separate each expression or statement.

  The variable and statement list following **PRINT#** in the above example consists of:

—a subscripted string variable:	C$(1,1)
—a string statement:	LEFT$(C$(1,1))
—a long integer variable:	A&
—an arithmetic expression:	A&/12
—an integer variable:	B%

NOTES

- Before transferring the data from the expressions to the files, **PRINT#** automatically performs any necessary numeric to string-type conversions, similar to the STR$ function.
- The use of commas instead of semicolons is not recommended because files have no tab positions. The SPC specification may be used instead.
- A ?# may replace the **PRINT#** keyword.

PRINT USING

TYPE Statement

FORMAT ? | **PRINT USING** *line number* | *string* | *string variable;* [*expression* [{, *expression*}]] [;]

ACTION Formats information output for screen display.

Formatted information is controlled within printing fields. Printing fields are defined by string format specifications. String format specifications must be enclosed in quotation marks. Like any other string, a string format specification may be assigned to a string variable.

EXAMPLE

PRINT USING "+.4#4E";1.12345
PRINT USING "+.####4E";1.12345
PRINT USING "#.4#4E";1.12345
PRINT USING "+.ZZZZ4E";1.12345

NOTES

- A question mark (?) may replace the PRINT keyword.
- String format specifications may consist of:

Numeric signs	+ *or* −
Dollar symbol	$
Characters	#, & *or* /
Letters	A, C, R, X *or* Z
Delimiters	, *or* ;
Repeat factor	(any positive integer from 1 to 255)

PRINT# USING

TYPE File statement

FORMAT ?# | **PRINT#** *file number* [, *record number*] **USING** *line number* | *string* | *string variable* [; *expression* [{, *expression*}]] [;]

ACTION Formats information output for screen display.

Formatted information is controlled within printing fields. Printing fields are defined by string format specifications. String format specifications must be enclosed in quotation marks. Like any other string, a string format specification may be assigned to a string variable.

EXAMPLE

PRINT# USING "+.4#4E";1.12345
PRINT# USING "+.####4E";1.12345
PRINT# USING "#.4#4E";1.12345
PRINT# USING "+.ZZZZ4E";1.12345

NOTES

- A ?# may replace the PRINT# keyword.
- String format specifications may consist of:

Numeric signs	+ *or* −
Dollar symbol	$
Characters	#, & *or* /
Letters	A, C, R, X *or* Z
Delimiters	, *or* ;
Repeat factor	(any positive integer from 1 to 255)

READ

TYPE Statement

FORMAT **READ** *variable* {, *variable*}]

ACTION Reads the data items (string or numeric) contained in a DATA statement and assigns them sequentially to the corresponding variables.

variable is a numeric, a string, or an array variable. The *variable* type must match the corresponding constant type in the DATA statement. The information contained in multiple DATA statements is read as if it were one continuous list. The **READ** statements access the DATA statements in line number order.

EXAMPLE

```
10  FOR D = 1 TO 3
20  READ X
30  PRINT X
40  NEXT D
50  DATA 10, 20, 30
```

RESULT

Line 10: Sets up a loop to repeat three times.

20: Reads a data item from the next DATA statement and assigns it to variable X.

30: Prints the contents of variable X on the screen.

40: Repeats from line 10.

50: DATA statement containing three items.

The printed result would be:

```
10
20
30
```

NOTES

- The **READ** and DATA statements work with both string and numeric variables. String constants in DATA statements do not need to be surrounded by quotation marks unless the string contains commas or blanks. DATA statements may be placed anywhere in the program.

READ#

TYPE File statement

FORMAT **READ#** *file number* [, *record number*][; *variable* [{, *variable*}]]

ACTION Reads data from a DATA file whose reference number is specified following the number sign.

READ# gets a line of data for each *variable* in its variable list.

EXAMPLE

```
READ#1, 32;A%,B&,C$
```

COMMENTS

- *record number* following *file number* specifies where reading should start.
- A comma separates *file number* from *record number. record number* is assigned to the first variable in the list.
- A semicolon must separate *record number* from the variable list. A comma must separate each *variable*.

  The variable list following **READ#** consists of:

A%	a real variable
B&	a long integer variable
C$	a string variable

NOTES

- **READ#** automatically performs any necessary type conversions for numeric data. However, type conversions are not automatically performed between numeric data and string variables (and vice versa).

REEC

stands for **RECORD**

TYPE File function

FORMAT **REC** (*file number*)

ACTION Returns the current record number of a specified file.

file number, enclosed in parentheses, can be any arithmetic expression.

EXAMPLE

REC(6)

NOTES

- If you use the INPUT# or READ# statements to access the catalog of a directory, **REC** returns the number of the line currently being accessed.
- Error messages displayed following nonvalid **REC** statements are: ?ILLEGAL QUANTITY ERROR, if the value of *record number* is not between 1 and 10; ?FILE NOT OPEN ERROR, if the specified file is not open.

REM

stands for **REMARK**

TYPE Statement

FORMAT **REM** *string*

ACTION Allows insertion of remarks or comments to document program.

string may be any sequence of characters.

EXAMPLE

```
10 REM *** The area of a circle is found by the formula :
20 REM *** C = PI * R^2 : PI = 3.14159265
30 REM *** Variables used :
40 REM *** C For Circle
50 REM *** R For Radius
60 REM *** Written on ......... By .........
70 INPUT R
80 C = 3.14159265 * R^2 : PRINT C
90 END
```

RESULT

Line 10–60: Remarks to document program.

70: Accepts input and assigns it to variable R.

80: Computes C and prints it on the screen.

NOTES

- The **REM** statements are not executed. Strings following **REM** need not be inside quotes. Any function or statement that follows **REM** on the same line or before a colon is ignored.
- If program execution branches to a **REM** statement from a GOTO or GOSUB statement, execution continues with the first executable statement after the **REM** statement.

RENAME

TYPE File statement

FORMAT **RENAME** *pathname1, pathname2*

ACTION Changes the names of volumes, subdirectories, or local files.

EXAMPLE

RENAME/Stock/Purchases/France, /Stock/Purchases/Foreign

RESULT

The statement above causes local file	France
to be renamed	Foreign
in the subdirectory	Purchases
stored in the disk whose volume name is	Stock.

NOTES

- **RENAME** cannot be used to create a file or subdirectory, only to rename an existing one. To create new files and root directories, you must use the CREATE statement.
- **RENAME** must be followed by *old pathname*, a comma, and *new pathname*.

RESTORE

TYPE Statement

FORMAT **RESTORE**

ACTION Allows the reuse of the same DATA by the READ statement.

After a **RESTORE** statement is executed, data associated with DATA statements can be reread, starting with the first item in the first DATA statement in the program.

EXAMPLE

```
10 FOR B = 1 TO 3
20 RESTORE
30 FOR D = 1 TO 3
40 READ X
50 PRINT X,
60 NEXT D
70 PRINT
80 NEXT B
90 DATA 10,20,30
```

RESULT

Line 10: Sets up a loop to repeat three times.

20: Allows READ statement to reread DATA.

30: Sets up a loop to read three times.

40: Reads the next data item; assigns it to the variable X.

50: Prints the value of X, suppressing line feed.

60: Repeats from line 30.

70: Outputs a line feed.

80: Repeats from line 10.

90: DATA statement with three items.

The printed result would be:

```
10    20    30
10    20    30
10    20    30
```

NOTES

- Each time the **RESTORE** statement is executed, the next READ statement begins with the first data item in the first DATA statement in the program.

RESUME

TYPE Statement

FORMAT **RESUME**

ACTION Resumes program execution at the beginning of the statement where an error has occurred.

EXAMPLE

```
10  ON ERR GOTO 100
20  INPUT "Enter any integer from 1 through 6"; X
30  IF N = 9 THEN END
40  A = 12/N
50  PRINT A : GOTO 20
    |
100 N = N + 1
110 RESUME
120 END
```

RESULT

Line 10: Error-trapping statement: if an error occurs, jumps to line 100.

20: Asks for input and assigns it to variable N.

30: Ends program execution if variable N is assigned a 9.

40: Divides 12 by N.

If variable N was assigned a 0 (zero), an error would occur, causing an unconditional jump to line 100.

50: Prints the value of A; jumps back to line 20.

100: Computes the new value of N.

110: Resumes program execution at line 40 where the "Division by Zero" error originally occurred.

NOTES

- ON ERR GOTO is used to avoid the display of the system's built-in error messages and the subsequent halting of the program execution, by jumping to an error-handling routine. **RESUME** is generally the last statement of the error-handling routine.

RETURN

TYPE Statement

FORMAT GOSUB *line number*

|

RETURN

ACTION Transfers program execution to the next executable statement after the last executed GOSUB statement.

GOSUB is used to set up subroutines that can be used more than once by various parts of the program. The subroutine consists of the statements between *line number* and **RETURN**. More than one GOSUB statement can be executed consecutively.

EXAMPLE

```
  10  PRINT "Type C to Continue, E to End."
  20  INPUT C$
  30  IF C$ = "C" THEN GOSUB 1000 : END
  40  IF C$ = "E" THEN END
  50  PRINT "Invalid entry. Try again." : GOTO 10
  60  END
      : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : :
1000  REM *** SUBROUTINE
      |
2000  RETURN
```

RESULT

Line 10: Prints the message.

20: Accepts input and assigns it to variable C$.

30: If C is typed, program execution passes to the subroutine at line 1000.

40: If E is typed, ends the program.

50: If any other character is typed, prints the message and jumps back to line 10.

60: Ends the program.

1000: Start of the subroutine.

2000: Returns program execution to the next statement following the most recently executed GOSUB statement.

NOTES

- A subroutine must always end with a **RETURN** statement to cause program execution to continue from the next statement following the GOSUB statement.

RIGHT$

TYPE String function

FORMAT PRINT **RIGHT$** (*string expression, number of characters*)

ACTION Returns the rightmost *number of characters* of *string expression.*

EXAMPLE

1. *string expression* can be a string constant;

   PRINT **RIGHT$** ("AFTERNOON",3) Returns OON
   PRINT **RIGHT$** ("AFTERNOON",4) Returns NOON

2. a string variable;

   A$ = "AFTERNOON"
   PRINT **RIGHT$** (A$,3) Returns OON
   PRINT **RIGHT$** (A$,4) Returns NOON

3. any valid combination thereof.

   A$ = "AFTER" : A = 4
   PRINT **RIGHT$** (A$ + "NOON",A) Returns NOON

NOTES

- If *number of characters* is greater than the total length of *string expression*, the entire string is returned.
- If *number of characters* = 0 (zero), the null string ("") is returned.
- The number of characters in a string expression may range from 0 (zero) to 255.
- A string variable is identified by a dollar sign ($).
- Business BASIC has 12 string or string-related functions: ASC, CHR$, HEX$, INSTR, LEFT$, LEN, MID$, **RIGHT$**, STR$, SUB$, TEN, VAL.

RND *stands for* **RANDOM**

TYPE Numeric function

FORMAT **RND** (*arithmetic expression*)

ACTION Returns a random number between 0 (zero) and 1.

arithmetic expression can be a numeric constant, a numeric variable, or an arithmetic operation. The returned sequence of random numbers varies depending on *arithmetic expression*'s value:

1. With a 0 (zero) as an argument value, **RND** returns a random real positive number less than 1.
2. With an argument value greater than 0 (zero), **RND** will return a different number each time it is used.
3. With a negative argument value, **RND** will return the same number each time the same argument is used.

Numeric functions may be used either in immediate mode in conjunction with a PRINT statement or in deferred execution. The argument to all numeric functions must be an *arithmetic expression*. All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions.

EXAMPLE

```
10  FOR J = 1 TO 5
20  PRINT RND (Ø)
30  NEXT J
40  END
```

RESULT

Line 10: Sets up a loop to repeat five times.

20: Prints a random number between 0 (zero) and 1.

30: Repeats from line 10.

NOTES

- Business BASIC has 16 numeric functions in the following type categories:

trigonometric:	ATN, COS, SIN, TAN
arithmetic:	ABS, EXP, INT, LOG, **RND**, SGN, SQR
conversion:	CONV, CONV%, CONV&, CONV$
user-defined:	DEF FN

RUN

TYPE Statement

FORMAT

1. **RUN** [*line number*]
2. **RUN** *file name*, [*line number*]

ACTION

1. Executes the current program stored in memory, beginning with *line number* if specified.
2. Loads and executes the program specified by *file name*, beginning with *line number* if specified.

EXAMPLE

```
10 INPUT Q$
20 IF Q$ = "YES" THEN RUN : END
30 RUN "PAYROLL"
40 END
```

RESULT

Line 10: Accepts input and assigns it to the string variable Q$.

20: If the string entered at line 10 is YES, program execution starts with the first line number (lowest).

30: Loads and runs the program PAYROLL if the string entered at line 10 is different from YES. Execution starts with the first line of the program.

NOTES

- **RUN** may be used in the immediate (command) mode by typing **RUN** and pressing the RETURN or ENTER key.
- If the line number specified after the statement **RUN** does not exist in the program, an error message is displayed.
- **RUN** reinitializes all numeric variables to 0 (zero) and string variables to null, clears all pointers and stacks, and closes all files.

SAVE

TYPE File statement

FORMAT **SAVE** *file name*

ACTION Writes a copy of the program currently in memory to a disk.

This copy is called a BASIC program file.

EXAMPLE

SAVE .D1/Inventory

COMMENTS

- The disk drive reference name consists of a period, the letter D, and the drive number. .D1 refers to the built-in disk drive. .D2, .D3, and .D4 will refer to additional external disk drives.
- *file name* must be preceded with a slash (/).

NOTES

- Saving a file on a disk that already contains a BASIC program with the same file name causes the erasure of the old file.
- If an error is made, the following messages are displayed: ?FILE LOCKED ERROR, if you try to save a file with the same file name as a locked BASIC program; ?TYPE MISMATCH ERROR, if you try to save a file with the same file name but which is not a BASIC program.

SCALE

TYPE Statement

FORMAT **SCALE** (*arithmetic expression*, *variable*)

ACTION Shifts the decimal point of a displayed value to the left or right of the original position.

arithmetic expression indicates the number of places and the direction in which the decimal point should be moved. *arithmetic expression* may be any positive or negative integer from −128 to 127. If *arithmetic expression* is positive, the decimal point is moved to the right. If negative, the decimal point is moved to the left. *variable* represents the actual numeric value to be displayed.

EXAMPLE

```
10  A& = 12345678901234567
20  PRINT USING "20&";SCALE(-3,A&)
```

RESULT

Line 10: Sets A& equal to the long integer value on the right of the equal sign.

20: Displays the value of A& according to the string format specification (20&) and the **SCALE** statement (−3,A&):

12,345,678,901,235

NOTES

- A **SCALE** statement may be used with a PRINT [#] USING statement.
- The resulting exponent of the value must be between −99 and +99, or an ?ILLEGAL QUANTITY ERROR occurs.

SGN

stands for **SIGN**

TYPE Numeric function

FORMAT **SGN** (*arithmetic expression*)

ACTION Returns the sign of *arithmetic expression*.

The function **SGN** is called the signum function. It returns −1 if the expression is negative; 0 (zero) if the expression is equal to 0 (zero); and 1 if the expression is positive.

Numeric functions may be used either in immediate mode in conjunction with a PRINT statement or in deferred execution. The argument to all numeric functions must be an *arithmetic expression.* All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions.

EXAMPLE

1. *arithmetic expression* can be a numeric constant;

PRINT **SGN** (0)	Returns 0
PRINT **SGN** (10)	Returns 1
PRINT **SGN** (−10)	Returns −1

2. a numeric variable;

A = 0 : B = 10 : C = −10	
PRINT **SGN** (A)	Returns 0
PRINT **SGN** (B)	Returns 1
PRINT **SGN** (C)	Returns −1

3. an arithmetic operation.

A = 0 : B = 10 : C = −10	
PRINT **SGN** (B + C)	Returns 0
PRINT **SGN** (A ^ 2 + B ^ 2)	Returns 1
PRINT **SGN** (A + (C ^ 2))	Returns −1

NOTES

- Business BASIC has 16 numeric functions in the following type categories:

trigonometric:	ATN, COS, SIN, TAN
arithmetic:	ABS, EXP, INT, LOG, RND, **SGN**, SQR
conversion:	CONV, CONV%, CONV&, CONV$
user-defined:	DEF FN

SIN

stands for **SINE**

TYPE Numeric function

FORMAT **SIN** (*arithmetic expression*)

ACTION Returns the sine of *arithmetic expression*.

Numeric functions may be used either in immediate mode in conjunction with a PRINT statement or in deferred execution. The argument to all numeric functions must be an *arithmetic expression.* All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions.

EXAMPLE

```
10 REM *** H = Hypotenuse of angle A
20 REM *** OS = Opposite side of angle A
30 REM *** A = Angle of a right triangle
40 FOR J = 1 TO 3
50 PRINT SIN (J)
60 NEXT J
70 END
```

RESULT

Line 10–30: Remarks to document program.

40: Sets up a loop to repeat three times.

50: Print the sine of J:

.841470985 for J = 1 (radian)
.909297427 for J = 2 (radians)
.141120008 for J = 3 (radians)

60: Repeats from line 40.

NOTES

- **SIN** is the opposite of ARCSIN. **SIN** (A) = OS/H
- *Conversions:*

  Radian = Degree / 57.29577951
  Degree = Radian * 57.29577951

- Business BASIC has 16 numeric functions in the following type categories:

trigonometric:	ATN, COS, **SIN**, TAN
arithmetic:	ABS, EXP, INT, LOG, RND, SGN, SQR
conversion:	CONV, CONV%, CONV&, CONV$
user-defined:	DEF FN

SPC *stands for* **SPACE**

TYPE Function

FORMAT **SPC** (*arithmetic expression*)

ACTION Inserts the requested number of spaces between two screen printing positions.

EXAMPLE

1. *arithmetic expression* can be a numeric constant;

PRINT "AB" **SPC**(5) "CD"	Inserts 5 spaces between the two strings: AB CD
PRINT "AB" **SPC**(7) "CD"	Inserts 7 spaces between the two strings: AB CD

2. a numeric variable;

A = 5 : B = 7

PRINT "AB" **SPC**(A) "CD"	Inserts 5 spaces between the two strings: AB CD
PRINT "AB" **SPC**(B) "CD"	Inserts 7 spaces between the two strings: AB CD

3. any valid combination thereof.

```
10 FOR J = 1 TO 4
20 PRINT " * " SPC(J) " * "
30 NEXT J
```

Inserts J spaces between the asterisks at each subsequent line:

```
*  *      J = 1
*   *     J = 2
*    *    J = 3
*     *   J = 4
```

NOTES

- The arithmetic expression must be in the range from 0 (zero) to 255.

SQR *stands for* **SQUARE ROOT**

TYPE Numeric function

FORMAT **SQR** (*arithmetic expression*)

ACTION Returns the square root of *arithmetic expression*.

Numeric functions may be used either in immediate mode in conjunction with a PRINT statement or in deferred execution. The argument to all numeric functions must be an *arithmetic expression.* All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions.

EXAMPLE

1. *arithmetic expression* can be a numeric constant;

PRINT **SQR** (0)	Returns 0
PRINT **SQR** (10)	Returns 3.16227766

2. a numeric variable;

A = 0 : B = 10	
PRINT **SQR** (A)	Returns 0
PRINT **SQR** (B)	Returns 3.16227766

3. an arithmetic operation.

A = 0 : B = 10	
PRINT **SQR** (A + (2 * 5) * B)	Returns 10
PRINT **SQR** (A ^ 2 + B ^ 2)	Returns 10
PRINT **SQR** (B ^ B)	Returns 100000

NOTES

- *arithmetic expression* must be positive.
- Business BASIC has 16 numeric functions in the following type categories:

trigonometric:	ATN, COS, SIN, TAN
arithmetic:	ABS, EXP, INT, LOG, RND, SGN, **SQR**
conversion:	CONV, CONV%, CONV&, CONV$
user-defined:	DEF FN

STEP

TYPE Clause

FORMAT FOR *control variable* = *aexpr1* TO *aexpr2* [**STEP** *aexpr3*]
|
NEXT [*control variable* {, *control variable*}]

ACTION FOR ... NEXT sets up a program loop that repeats the series of instructions inside the loop a given number of times.

aexpr is an *arithmetic expression.* The loop begins with the FOR statement and ends with the NEXT statement. Every statement in between is executed once with each repetition. Every repetition automatically increments (adds to) the value of *control variable* by a value equal to *aexpr3*; if **STEP** is omitted, the default increment is 1. *control variable* starts off having a value equal to *aexpr1*; when the value of *control variable* reaches *aexpr2*, the loop is ended and program execution continues with the statement after NEXT. A conditional statement can be
used to exit the loop before it is finished.

EXAMPLE

```
10  FOR B = 10 TO 140 STEP 10
20  PRINT "AZ";
30  NEXT B
40  END
```

RESULT

Line 10: Sets up a loop to repeat 14 times.

20: Prints the string AZ.

30: Repeats from line 10.

NOTES

- The initial value of *control variable* B has been incremented by the **STEP** value of 10.
- A loop structure may contain other loops within it, provided that the loops are nested.

STOP

TYPE Statement

FORMAT **STOP**

ACTION Halts program execution and returns to command (keyboard) level.

EXAMPLE

```
10 PRINT "This program starts at line number 10"
20 STOP
30 PRINT "Execution continues with this statement"
```

RESULT

Line 10: Prints the string:

This program starts at line number 10

20: The **STOP** statement temporarily halts program execution and causes the following message to be displayed:

BREAK IN 20

(that is, in line number 20)

Typing CONT on the keyboard and pressing the RETURN key causes execution to continue with the next instruction following the **STOP** statement.

30: Prints the string:

Execution continues with this statement

NOTES

- Program execution can also be temporarily halted by pressing the CONTROL key followed by the letter C. Unlike the END statement, the **STOP** statement does not close files.
- **STOP** statements may be used anywhere in a program.

STR$ *stands for* **STRING**

TYPE String function

FORMAT **STR$** (*arithmetic expression*)

ACTION Returns a representation of *arithmetic expression* in string form.

If *arithmetic expression* is positive, the returned string contains a leading blank—the space reserved for the plus (+) sign.

EXAMPLE

1. *arithmetic expression* can be a numeric constant;

   PRINT **STR$** (12345) — Returns 12345
   PRINT **STR$** (123.45) — Returns 123.45

2. a numeric variable;

   A = 12345 : B = 123.45
   PRINT **STR$** (A) — Returns 12345
   PRINT **STR$** (B) — Returns 123.45

3. any valid combination thereof.

   A = 12345 : B = 123.45
   PRINT **STR$** (A + B) — Returns 12468.45

NOTES

- Conversion of an arithmetic expression into a string expression permits manipulation by the available string functions.

  Example:

  A = 123456789 : A$ = **STR$** (A)

  PRINT LEFT$ (A$,4) — Returns 1234
  PRINT RIGHT$ (A$,5) — Returns 56789

- Business BASIC has 12 string or string-related functions: ASC, CHR$, HEX$, INSTR, LEFT$, LEN, MID$, RIGHT$, **STR$**, SUB$, TEN, VAL.

SUB$ *stands for* **SUBSTRING**

TYPE String function

FORMAT **SUB$** (*string expression*, *arithmetic expression* [, *arithmetic expression*]) = *string expression*

ACTION Replaces any part of a string expression with a substring starting at a specified position.

string expression may be a string constant or a string variable.

EXAMPLE

```
10 A$ = "ARITHMETIC EXPRESSIONS"
20 B$ = "COMPUTATION"
30 SUB$ (A$,12) = B$
40 PRINT A$
50 END
```

RESULT

Line 10: A string is assigned to string variable A$.

20: A substring is assigned to string variable B$.

30: Replaces part of string expression A$ starting at character position 12 by substring B$.

40: Prints the new value of the string variable A$:

ARITHMETIC COMPUTATION

NOTES

- The dollar sign ($) is an identifier that defines a function or a variable name as being of the string type.
- You may optionally include a second arithmetic expression to specify the number of characters in the substring to replace characters in the original string.
- Business BASIC has 12 string or string-related functions: ASC, CHR$, HEX$, INSTR, LEFT$, LEN, MID$, RIGHT$, STR$, **SUB$**, TEN, VAL.

SUBTRACTION *symbol* −

TYPE Arithmetic operator

FORMAT *numeric expression1* − *numeric expression2*

ACTION Performs an arithmetic subtraction.

EXAMPLE

1. *numeric expression* can be a numeric constant;

PRINT 20 − 10	Returns 10
PRINT 20 − 10 − 5	Returns 5
PRINT 20 − 25	Returns −5

2. a numeric variable;

A = 20 : B = 10 : C = 5 : D = 25	
PRINT A − B	Returns 10
PRINT A − B − C	Returns 5
PRINT A − D	Returns −5

3. any valid combination thereof.

A = 20 : B = 10 : C = 5 : D = 25	
PRINT A − 10	Returns 10
PRINT 20 − B − C	Returns 5
PRINT A − D	Returns −5

NOTES

- Business BASIC has 9 arithmetic operators:

+	Unary plus
−	Unary minus
^	Exponentiation
*	Multiplication
/	Floating-point division
MOD	Modulo division
DIV	Integer division
+	Addition
−	Subtraction

SWAP

TYPE Statement

FORMAT **SWAP** *variable1*, *variable2*

ACTION Exchanges the values of two variables of the same type.

Any type of variable may be SWAPped (real, integer, long integer, string), but the two variables must be of the same type.

EXAMPLE

```
10 READ X,Y
20 PRINT X,Y
30 IF X < Y THEN SWAP X,Y
40 PRINT X,Y
50 DATA 4,7
60 END
```

RESULT

Line 10: Reads and assigns the DATA values 4 and 7 to the variables X and Y, respectively.

20: Prints the values: 4   7.

30: The condition being true (X = 4 is smaller than Y = 7), the "swapping" of the two values will be executed: 4 will be stored in variable Y, and 7 will be stored in variable X.

40: Prints the new values: 7   4.

NOTES

- The **SWAP** statement is very useful in sorting operations.

TAB

stands for **TABULATOR**

TYPE Function

FORMAT **TAB** (*arithmetic expression*)

ACTION Spaces to the specified absolute position from the leftmost printing position.

arithmetic expression must be in the range from 1 to 255. (1 is the leftmost printing position on the screen.) If the current printing position is already beyond *arithmetic expression*, **TAB** is ignored.

EXAMPLE

1. *arithmetic expression* can be a numeric constant;

PRINT **TAB** (5) "AB"	Spaces to the fifth position before printing AB
PRINT **TAB** (7) "AB"	Spaces to the seventh position before printing AB

2. a numeric variable;

A = 5 : B = 7

PRINT **TAB** (A) "AB"	Spaces to the fifth position before printing AB
PRINT **TAB** (B) "AB"	Spaces to the seventh position before printing AB

3. any valid combination thereof.

```
10 FOR J = 1 TO 4
20 PRINT TAB (J) " *"
30 NEXT J
```

Spaces to the Jth position before printing the asterisk at each subsequent line:

```
*       J = 1
 *      J = 2
  *     J = 3
   *    J = 4
```

NOTES

- The **TAB** function is generally used with the PRINT statement to line up information in columns.

TAN

stands for **TANGENT**

TYPE Numeric function

FORMAT **TAN** (*arithmetic expression*)

ACTION Returns the tangent of *arithmetic expression*.

Numeric functions may be used either in immediate mode in conjunction with a PRINT statement or in deferred execution. The argument to all numeric functions must be an *arithmetic expression.* All floating-point arithmetic in Business BASIC is done with 32-bit precision, and this sets limits on the accuracy of the results returned by numeric functions.

EXAMPLE

```
10  REM *** OS = Side opposite to angle A
20  REM *** AS = Side adjacent to angle A
30  REM *** A = Angle of a right triangle
40  FOR J = 1 TO 3
50  PRINT TAN (J)
60  NEXT J
70  END
```

RESULT

Line 30: Remarks to document program.

40: Sets up a loop to repeat three times.

50: Prints the tangent of J:

1.55740772 for J = 1 (radian)
−.218503987 for J = 2 (radians)
−.142546543 for J = 3 (radians)

60: Repeats from line 40.

NOTES

- ARCTAN is the opposite of **TAN**. **TAN** (A) = OS/AS
- *Conversions:*

  Radian = Degree / 57.29577951
  Degree = Radian * 57.29577951
- Business BASIC has 16 numeric functions in the following type categories:

trigonometric:	ATN, COS, SIN, **TAN**
arithmetic:	ABS, EXP, INT, LOG, RND, SGN, SQR
conversion:	CONV, CONV%, CONV&, CONV$
user-defined:	DEF FN

TEN

TYPE String function

FORMAT **TEN** (*string expression*)

ACTION Returns the decimal equivalent of a hexadecimal value.

The last four characters of *string expression* must represent a hexadecimal value.

Example: PRINT **TEN**("conversion of the hexadecimal value CCCC")

will return: −13108, the *last four* characters CCCC representing a hexadecimal value.

EXAMPLE

```
10  DIM H$ (15)
20  FOR J = 1 TO 15
30  READ H$ (J)
40  PRINT TEN (H$(J)),
50  NEXT J
60  DATA "ØØØ1", "ØØØ2", "ØØØ3", "ØØØ4", "ØØØ5", "ØØØ6"
70  DATA "ØØØ7", ""ØØØ8", "ØØØ9", "ØØØA", "ØØØB", "ØØØC"
80  DATA "ØØØD", "ØØØE", "ØØØF"
90  END
```

RESULT

Line 10: Dimensions a list of 15 elements.

20: Sets up a loop to repeat 15 times.

30: Reads 15 data items.

40: Prints the decimal equivalent of hexadecimal values:

1	2	3	4	5	6
7	8	9	10	11	12

50: Repeats from line 20.

NOTES

- The returned decimal value is in the range from −32768 to +32767.
- Business BASIC has 12 string or string-related functions: ASC, CHR$, HEX$, INSTR, LEFT$, LEN, MID$, RIGHT$, STR$, SUB$, **TEN**, VAL.

THEN

TYPE Statement

FORMAT IF *logical expression* **THEN** *line number* | *statement list* [: ELSE *line number* | *statement list*]

ACTION Sends program execution to *line number* or executes *statement list* following **THEN** if *logical expression* is true (non-zero); otherwise:

1. if no ELSE clause is used, program execution passes to the next line in sequence;
2. if the ELSE clause is used, program execution passes to *line number* or *statement list* following ELSE.

IF ... **THEN** is called a conditional statement; it is one of the most commonly used statements in BASIC. It redirects program execution on the basis of the truth or falsity of *logical expression*. *logical expression* is usually a relational expression, comparing two values with relational operators.

EXAMPLE

```
10  INPUT "YES OR NO";X$
20  IF X$ = "YES" THEN 40
30  IF X$ = "NO" THEN 50 : ELSE 10
40  PRINT "Program execution is transferred to line 40" : END
50  PRINT "Program execution is transferred to line 50"
```

RESULT

Line 10: Asks for input; assigns the response to the variable X$.

20: If X$ is YES, program execution jumps to line 40.

30: If X$ is NO, execution jumps to line 50; otherwise, the statement following ELSE is executed.

40: Prints the message. Ends the program.

50: Prints the message.

NOTES

- The ELSE clause cannot be on a separate program line.

TO

TYPE Statement

FORMAT FOR *control variable* = *aexpr1* **TO** *aexpr2* [STEP *aexpr3*]
|
NEXT [*control variable* {, *control variable*}]

ACTION Sets up a program loop that repeats the series of instructions inside the loop a given number of times.

aexpr is an *arithmetic expression.* The loop begins with the FOR statement and ends with the NEXT statement. Every instruction in between is executed once with each repetition. Every repetition automatically increments (adds to) the value of *control variable* by a value equal to *aexpr3*; if STEP is omitted, the default increment is 1. *control variable* starts off having a value equal to *aexpr1*; when the value of *control variable* reaches *aexpr2*, the loop is ended and program execution continues with the statement after NEXT. A conditional statement can be used to exit the loop before it is finished.

EXAMPLE

```
10  FOR B = 1 TO 10
20  PRINT "AZ";
30  NEXT B
40  END
```

RESULT

Line 10: Sets up a loop to repeat 10 times.

20: Prints the string AZ.

30: Repeats from line 10.

NOTES

- The initial value of *control variable* B has been incremented by the default value of 1.
- A loop structure may contain other loops within it, provided that the loops are nested.

TRACE

TYPE Command

FORMAT **TRACE**

ACTION Used mainly to debug (check, troubleshoot) the sequential execution of a program or parts of it.

During program execution, **TRACE** displays a number sign (#) followed by the line numbers of the statements in the sequential order of their execution. Assignment statements are reported only by their line numbers. When a PRINT statement is encountered, **TRACE** displays the line number and the result of the PRINT statement.

EXAMPLE

```
10  A = 25
20  B = 55
30  C = A + B
40  PRINT C
```

TRACE

RUN

RESULT

#10 #20 #40 80

NOTES

- **TRACE** may be used either in the immediate (command) mode or in the deferred mode.
- Traced execution of assignment statements is denoted only by the statements' line numbers. If the traced statement contains a PRINT statement, **TRACE** displays the line number and the result of the PRINT statement.
- **TRACE** is switched off by "rebooting," LOAD *pathname*, RUN *pathname*, or by typing NOTRACE. The RUN command/statement not followed by a *pathname* or CHAIN does not cancel **TRACE**.

TYP

stands for **TYPE (OF DATA)**

TYPE File function

FORMAT **TYP** (*file number*)

ACTION Determines what type of data will be read from a particular file on the next access to that file.

file number, enclosed in parentheses, can be any arithmetic expression.

EXAMPLE

1. **TYP** (6)
2. ON **TYP** (6) GOSUB 1000, 2000, 3000, 4000, 5000

RESULT

1. The number returned by the **TYP** function denotes what type of data will next be read from the file whose *file number* is 6.
2. Depending on the number returned by the **TYP** function, program execution will branch to one of the line numbers following the GOSUB statement.

NOTES

- For a DATA file, **TYP** returns the following numbers:

1	For real
2	For integer
3	For long integer
4	For string
Ø	Indicates that the file is indeterminate

  For a TEXT file, **TYP** returns the value 8. Number 5 indicates an end-of-file marker, whether it is a data or text type file.
- Error messages displayed with nonvalid **TYP** statements are: ?ILLEGAL QUANTITY ERROR, if *file number* is not between 1 and 10; ?FILE NOT OPEN ERROR, if the specified file is not open.

UNLOCK

TYPE File statement

FORMAT **UNLOCK** *pathname*

ACTION Unlocks files previously protected (locked) by a LOCK statement.

A locked file may again be deleted, changed, renamed, or saved after it is unlocked by the **UNLOCK** statement. **UNLOCK** must be followed by the file or subdirectory name you wish to unlock.

EXAMPLE

UNLOCK/Purchases/Suppliers/France

NOTES

- When listed by a CATALOG statement, unlocked files are shown without the asterisk (*) that had previously appeared to the left of their file type, after a LOCK command has been executed.

Type	*Blks*	*Name*
BASIC	*00003*	*TRANSACTIONS*
DATA	*00015*	*PHONE.NUMBERS*
FOTO	*00009*	*STATISTICS*

VAL

stands for **VALUE**

TYPE String function

FORMAT **VAL** (*string expression*)

ACTION Returns the numerical value of *string expression*.

string expression should evaluate to a string representing a number. **VAL** converts the string into the number it represents. If *string expression* is not numeric, **VAL** will return a 0 (zero).

EXAMPLE

1. *string expression* can be a string constant;

   PRINT **VAL** ("12345") — Returns 12345
   PRINT **VAL** ("123.45") — Returns 123.45

2. a string variable;

   A$ = "12345" : B$ = "123.45"
   PRINT **VAL** (A$) — Returns 12345
   PRINT **VAL** (B$) — Returns 123.45

3. any valid combination thereof.

   A$ = "123" : B$ = ".45"
   PRINT **VAL** (A$ + B$) — Returns 123.45

NOTES

- Conversion of a string expression into a numeric expression permits subsequent arithmetic operations.

  Example:

  A$ = "123.45" : A = **VAL** (A$)
  PRINT INT (A) — Returns 123
  PRINT SIN (INT(A)) — Returns −.459903491

- Business BASIC has 12 string or string-related functions: ASC, CHR$, HEX$, INSTR, LEFT$, LEN, MID$, RIGHT$, STR$, SUB$, TEN, **VAL**.

VPOS

stands for **VERTICAL** *and* **POSITION**

TYPE Reserved variable

FORMAT **VPOS** = *arithmetic expression*

ACTION Specifies the vertical position of the cursor within a "window" or total screen.

A PRINT **VPOS** statement returns the current vertical position of the cursor. The position is relative to the upper margin of the window or total screen. *arithmetic expression* can be any integer constant or variable or any real arithmetic expression.

EXAMPLE

VPOS = 6

moves the cursor vertically to the sixth line within the current window.

NOTES

- All parameters are relative to the current window dimensions. For instance, in **VPOS** = 1, 1 specifies the first line within the current window.
- When **VPOS** is used to move the cursor vertically, the cursor's horizontal position is not affected.
- Values must be within the range from 0 (zero) to 255. A value of 0 (zero) is automatically converted to a value of 1. **VPOS** cannot move the cursor to a position outside of the window. **VPOS** values greater than the height of the window cause the cursor to move to the bottom line of the window.

WINDOW

TYPE Statement

FORMAT **WINDOW** *aexpres1*, *aexpres2* TO *aexpres3*, *aexpres4*

ACTION Sets the position and size of the "window" (any square or rectangle area within the total screen) where text is displayed.

aexpres is an *arithmetic expression* specified by a numeric constant, a numeric variable, or an arithmetic computation. *aexpres1* and *aexpres2* specify the upper-left corner. *aexpres3* and *aexpres4* following the word TO specify the lower-right corner of the window.

EXAMPLE

100 **WINDOW** 6,9 TO 16,19

COMMENTS

- 6 is the horizontal coordinate (column 6).
- 9 is the vertical coordinate (row 9) of the upper-left corner of the window.
- 16 is the horizontal coordinate (column 16).
- 19 is the vertical coordinate (row 19) of the lower-right corner of the window.

NOTES

- When a **WINDOW** statement is executed, the cursor moves to the lower-left corner of the specified window. (The HOME command moves it to the upper-left corner.)
- A coordinate value of 0 (zero) is automatically converted to a value of 1. Each value must be within the range from 0 (zero) to 255.
- The parameter values are relative to the limits of the screen. The size of the window cannot exceed that of the screen, namely, 80 columns by 24 lines.

WRITE#

TYPE File statement

FORMAT **WRITE#** *file number* [, *record number*] [; *expression* [{, *expression*}]]

ACTION Writes sequentially the value of each expression in its expression list to a field in a data file whose reference number is specified following the number sign.

WRITE# writes one line of data for each expression in the expression list.

EXAMPLE

WRITE#1,32;A%,B&,C$

COMMENTS

- *record number* following *file number* specifies where writing should start. The value of the first *expression* is written to the first field in the specified record. If no *record number* is specified, records are written sequentially.
- A comma separates *file number* from *record number*.
- A semicolon must separate *record number* from the variable list.
- A comma must separate each *expression*.

NOTES

- **WRITE#** performs no numeric to string-type conversions while transferring information from *expressions* to the file; it just writes a binary image of numeric data to the file.
- An integer is written as an integer only if an integer variable is specified. If the integer is part of an arithmetic expression, the expression value will be written as a real number.

Index of Symbols

The following is an index of valid symbols and their references in the guide.

FOR		SEE
Arithmetic Operators		
+	Plus sign	**ADDITION**
/	Slash	**DIVISION**
^	Caret	**EXPONENTIATION**
*	Asterisk	**MULTIPLICATION**
( )	Parentheses	**PARENTHESES**
–	Minus sign	**SUBTRACTION**
Delimiters		
:	Colon	**COLON**
,	Comma	**PRINT**
;	Semicolon	**PRINT**
Identifiers		
&	Long integer type	**AMPERSAND**
$	String type	**DOLLAR**
%	Integer type	**PERCENT**
Relational Operators		
=	Equal sign	**EQUAL TO**
>	Greater than sign	**GREATER THAN**
>= *or* =>	Greater than or equal to	**GREATER THAN OR EQUAL TO**
<	Less than sign	**LESS THAN**
<= *or* =<	Less than or equal to	**LESS THAN OR EQUAL TO**
<> *or* ><	Not equal to	**NOT EQUAL TO**
Miscellaneous		
=	Equal sign	**ASSIGNMENT**
+	Plus sign	**CONCATENATION**
?	Question mark	**PRINT**

Index of Keywords by Function

The following is an index of all keywords in the guide grouped by function.

Identifiers

Ampersand
Dollar
Percent

Input Statements

DATA
GET
INPUT
READ
RESTORE

Logical Operators

AND
NOT
OR

Loop Statements

FOR
NEXT
STEP
TO

Relational Operators

Equal To
Greater Than
Greater Than or Equal To
Less Than
Less Than or Equal To
Not Equal To

Remark Statement

REM

Reserved Variables

EOF
ERR
ERRLIN
FRE
HPOS
INDENT
KBD
OUTREC
PREFIX$
VPOS

Screen Statements

DEL
HOME
INVERSE
LIST
NORMAL
PRINT
SPC
TAB
WINDOW

String and String-Related Functions

ASC
CHR$
Concatenation
HEX$
INSTR
LEFT$
LEN
MID$
RIGHT$
STR$
SUB$
TEN
VAL

System and Utility Statements

CHAIN
CLEAR
CONT
END
LOAD

System and Utility Statements
(*continued*)

NEW
RUN
SAVE
STOP

Trigonometric Functions

ATN
COS
SIN
TAN

Type Conversion Functions

CONV
CONV%
CONV&
CONV$

Unconditional Branching Statements

GOSUB
GOTO
POP
RETURN

User-Defined Function

DEF FN

Computers $14.95

A reference that belongs next to every Apple® *III* keyboard...

BASIC KEYWORDS FOR THE APPLE® *III*

This complete, easy-to-use dictionary lists and defines the BASIC vocabulary for the Apple *III*—statements, commands, functions, operators, symbols... everything! And it explains how they're used in popular business applications.

Open to any page... *Basic Keywords* is organized just like any dictionary, but with *only one word per page.* Every entry includes the keyword and what it stands for... its type (command or statement)...and its action—that is, its general purpose and how it operates. Program and syntax examples, together with insightful comments and notes, highlight the keyword's essential characteristics and help you understand its function and use.

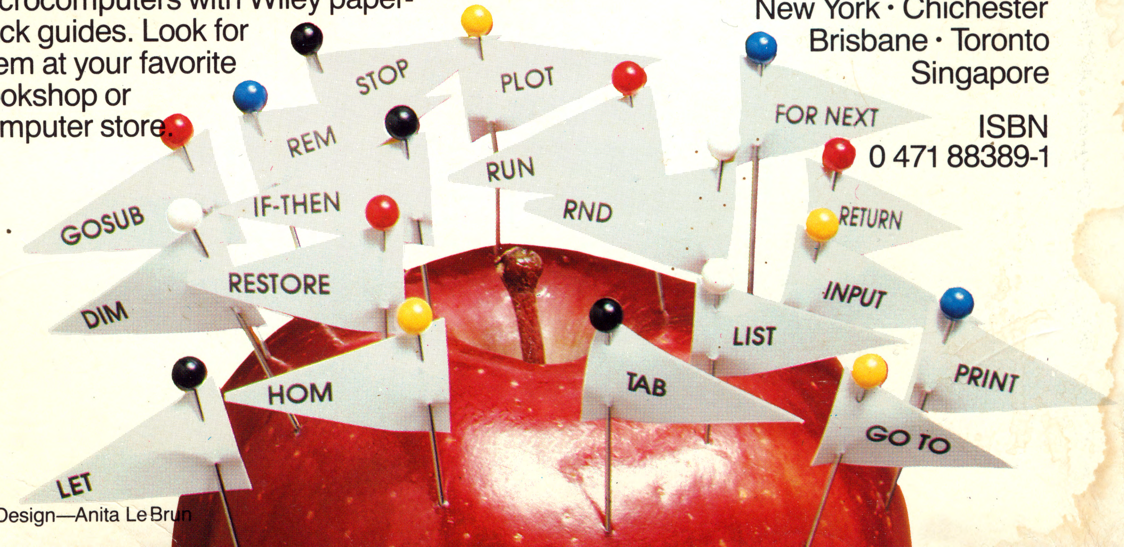
Need more help using your Apple *III*? Every definition in *Basic Keywords for the Apple III* is cross-referenced to the author's popular Self-Teaching Guide, *Business BASIC for the Apple III.*® Here you'll find detailed, step-by-step guidance for all your Apple *III* programming needs.

EDDIE ADAMIS is a software consultant in France. A computer hobbyist and programmer for many years, his articles appear regularly in computer publications in France. He is the author of *BASIC Subroutines for Commodore Computers, BASIC Keywords: A User's Reference, Business BASIC for the Apple® III, and Business BASIC for the IBM PC.*

WILEY PRESS, a division of
JOHN WILEY & SONS, Inc.
605 Third Avenue
New York, N.Y. 10158
New York • Chichester
Brisbane • Toronto
Singapore

ISBN
0 471 88389-1

More than two million people have learned to program, use, and enjoy microcomputers with Wiley paperback guides. Look for them at your favorite bookshop or computer store.



Cover Design—Anita Le Brun