MICROSOFT

Z-Basic (Z-DOS™)

Volume II

593-0050-01
CONSISTS OF

MANUAL
595-2834-01
FLYSHEET
597-2866-01

Printed in the
United States of America

ZENITH data systems

HEATH

NOTICE

This software is licensed (not sold). It is licensed to sublicensees, including end-users, without either express or implied warranties of any kind on an "as is" basis.

The owner and distributors make no express or implied warranties to sublicensees, including end-users, with regard to this software, including merchantability, fitness for any purpose or non-infringement of patents, copyrights or other proprietary rights of others. Neither of them shall have any liability or responsibility to sublicensees, including end-users, for damages of any kind, including special, indirect or consequential damages, arising out of or resulting from any program, services or materials made available hereunder or the use or modification thereof.

Technical consultation is available for any problems you encounter in verifying the proper operation of these products. Sorry, but we are not able to evaluate or assist in the debugging of any programs you may develop. For technical assistance, call:

(616) 982-3884 Application Software/SoftStuff Products
(616) 982-3860 Operating System/Language Software/Utilities

Consultation is available from 8:00 AM to 4:30 PM (Eastern Time Zone) on regular business days.

Zenith Data Systems Corporation
Software Consultation
Hilltop Road
St. Joseph, Michigan 49085

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

ZENITH DATA SYSTEMS CORPORATION
ST. JOSEPH, MICHIGAN 49085

ABS Function

BRIEF

Format: `ABS(X)`

Action: Returns the absolute value of the expression X.

Details

The ABS function returns the absolute value of X without regarding the sign of X. Given a positive value, it returns that value. Given a negative value, it returns the corresponding positive value.

Example:

```
PRINT ABS (7*(-5))
 35
Ok
```

ASC Function

BRIEF

Format: `ASC(X$)`

Action: Returns a numerical value that is the ASCII code of the first character of the string X$.

Details

The system used to represent characters is called ASCII (American Standard Code for Information Interchange). There are 128 possible characters that correspond to 128 seven-bit codes in the ASCII character set. In BASIC you have the option of interpreting the seven-bit patterns as the decimal equivalents.

The job of converting between these two interpretations is performed by the ASC and CHR$ functions. CHR$ is covered on Page 10.15. ASC returns the decimal equivalent of the first character in the string acting as the argument. The ASC function can only operate on single characters since (like all functions) it can only return a single result.

If X$ is null, an `Illegal Function Call` error message is returned. (See Appendix C for ASCII codes.)

Example:

```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
```

See the CHR$ function Page 10.15 for ASCII-to-string conversion.

ATN Function

BRIEF

Format: `ATN(X)`

Action: Returns the arctangent of X in radians.

Details

The ATN function (arctangent) is the inverse function of the tangent (TAN). If Y is the tangent of zero, then zero is the arctangent of Y. The result of the ATN function is in the range −PI/2 to PI/2 where PI=3.14159.

The expression X may be any numeric type, but the evaluation of ATN is always performed in single precision.

Example:

```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
 1.249046
Ok
```

ALPHABETICAL REFERENCE GUIDE

AUTO Command

BRIEF

Format: `AUTO [<line number>[,< increment>]]`

Purpose: To generate a line number automatically after every RETURN.

Details

AUTO begins numbering at <line number> and increases each subsequent line number by <increment>. If both the line number or the increment value is unspecified, the assumed value (default value) for both line number and increment is 10. If the line number is followed by a comma but the increment is not specified, the last increment specified in an AUTO command is assumed. If line number is unspecified and the increment value is specified, the starting line number defaults to zero.

Asterisk Warning

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn you that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing **CTRL-C**. The line in which CTRL-C is typed is not saved. After CTRL-C is typed, BASIC returns to command level, which means you must type **AUTO** again to generate line numbers automatically.

Examples:

`AUTO 100,50`	Generates line numbers 100, 150, 200 ...
`AUTO`	Generates line numbers 10, 20, 30, 40...
`AUTO 60`	Will start with line 60 and increment subsequent lines using the default increment value of 10.
`AUTO, 60`	Will start at 0 and increment subsequent lines using the increment value of 50.

For information on editing program lines, see Chapter 3 Page 3.6.

BEEP Statement

BRIEF

Format: `BEEP`

Purpose: The BEEP statement sounds the speaker at 1000 Hz for 1/4 second.

Details

Non-graphic versions of BASIC use PRINT CHR$(7) to send an ASCII bell character. Both BEEP and PRINT CHR$(7) have the same effect.

The BEEP statement can be used in a variety of applications. It can be incorporated into a game as a signal for some type of response, or it can be used as an error trapping signal as shown below.

Example:

```
2420 REM If X is out of range, complain in line 2430.
2430 IF X< 20 THEN BEEP
```

BLOAD Command

BRIEF

Format: `BLOAD <file spec> [,<offset>]`

Purpose: The BLOAD statement allows a file to be loaded anywhere in user memory.

Details

File spec Is a valid string expression containing the device and file name. The file name may be one to eight characters in length.

Offset Is a valid numeric expression returning an unsigned integer in the range zero to 65535. This is the offset into the segment declared by the last DEF SEG statement.

BLOAD and BSAVE are most useful for loading and saving machine language programs. (See "CALL Statement"). However, BLOAD and BSAVE are not restricted to only machine language programs. Any segment may be specified as the source or target for these statements via the DEF SEG statement. BLOAD and BSAVE provide a convenient way of saving and displaying graphic images.

CTRL-C may be typed at any time during BLOAD or LOAD. If it is used between files or after a time-out period, BASIC will exit the search and return to direct mode. Previous memory contents remain unchanged.

If the BLOAD command is executed in a BASIC program, the filenames skipped and found are not displayed on the screen.

Rules:

1. If the device identifier is omitted and the filename is less than one character or greater than eight characters in length, a `Bad File Name` error is issued and the load is aborted.

2. If an offset is omitted, the offset specified at BSAVE is assumed. That is, the file is loaded into the same location it was saved from.

3. If an offset is specified, a DEF SEG statement should be executed before the BLOAD. When offset is given, BASIC assumes you want to BLOAD at an address other than the one saved. The last known DEF SEG address will be used.

4. **CAUTION:** BLOAD does not perform an address range check. It is possible to BLOAD anywhere in memory. You must not BLOAD over BASIC stack, BASIC Program, or BASIC's variable area.

Example:

```
10 "Load a machine language program into memory at 60:F000
20 DEF SEG 'Restore Segment to BASIC DS.
30 BLOAD"PROG1",&HE000

10 'Load the screen
20 DEF SEG= &HE000 'Point segment at green plane.
30 BLOAD "PICTURE" ,0 'Load file PICTURE into green plane.
```

Note the DEF SEG statement in line 20 and the offset of zero in line 30. This guarantees that the correct address is used.

The BSAVE example on Page 10.9 illustrates how "PICTURE" was saved.

BSAVE Command

BRIEF

Format: `BSAVE <file spec>,<offset>,<length>`

Purpose: Allows portions of memory to be written and saved to the specified device.

Details

Filespec Is a valid string expression containing the device and file name. The file name may be one to eight characters in length.

Offset Is a valid numeric expression returning an unsigned integer in the range zero to 65535. This is the offset into the segment declared by the last DEF SEG to start saving from.

Length Is a valid numeric expression returning an unsigned integer in the range one to 65535. This is the length of the memory image to be saved.

BLOAD and BSAVE are most useful for loading and saving machine language programs. (See "CALL Statement"). However, BLOAD and BSAVE are not restricted to only machine language programs. Any segment may be specified as the source or target for these statements via the DEF SEG statement. BLOAD and BSAVE provide a convenient way of saving and displaying graphic images.

Rules:

1. If filename is less than one character, or greater than eight characters in length, a `Bad File Name` error is issued and the save aborted.

2. If offset is omitted, a Syntax error message is issued and the save aborted. A DEF SEG statement should be executed before the BSAVE. The last known DEF SEG address is always used for the save.

3. If length is omitted, a Syntax error message is issued and the save aborted.

BSAVE Command

Example:

```
10 'Save the green plane.
20 'Point segment at green plane.
30 DEF SEG= &HE000
40 'Save green plane in file PICTURE.
50 BSAVE "PICTURE",0,&H8000
```

The DEF SEG statement must be used to set the segment address to the start of the screen buffer. Offset of zero and length &H8000 specifies that the entire 32K screen buffer is to be saved.

CALL Statement

BRIEF

Format: `CALL <variable name> [ ( <argument list> ) ]`

Purpose: To call an assembly language subroutine.

<variable name> contains the address that is the starting point in memory of the subroutine being called.

<argument list> contains the variables or constants, separated by commas, that are to be passed to the routine.

Details

The CALL statement is the recommended way of interfacing 8086 assembly language programs with Z-BASIC. It is further suggested that the old style user call (x=USR(n)) not be used.

Invocation of the CALL statement causes the following to occur:

1. For each parameter in the argument list, the two byte offset of the parameter's location within the data segment (DS) is pushed onto the stack.

2. BASIC's return address code segment (CS) and offset are pushed onto the stack.

3. Control is transferred to the user's routine via an 8086 long call to the segment address given in the last DEF SEG statement and offset given in <variable name>.

Example:

```
100 DEF SEG=&H8000
110 FOO=0
120 CALL FOO(A,B$,C)
        .
        .
        .
```

In the preceding program, line 100 sets the segment to 8000 Hex. F00 is set to zero so that the call to F00 will execute the subroutine at location Hex 8000H.

The following sequence of 8086 assembly language demonstrates access of the parameters passed. Storing a return results in the variable 'C'.

```
MOV   BP,SP       ;Get current Stack posn in BP.
MOV   BX,6[BP]    ;Get address of B$ dope.
MOV   CL,[BX]     ;Get length of B$ in CL.
MOV   DX,1[BX]    ;Get addr of B$ text in DX.
      .
      .
      .
MOV   SI,8[BP]    ;Get address of 'A' in SI.
MOV   DI,4[BP]    ;Get pointer to 'C' in DI.
MOVS  WORD        ;Store variable 'A' in 'C'.
RET   6           ;Restore Stack, return.
```

Note that, the called program must know the variable type for numeric parameters passed. In the above example, the instruction MOVS WORD will copy only two bytes. This is fine if variables A and C are integers. We would have to copy four bytes if they were single precision and copy eight bytes if they were double precision.

For a more detailed explanation of this command see Appendix E, "Assembly Language Subroutines".

The CALL statement conforms to the INTEL PL/M-86 "Calling Conventions" outlined in Chapter 9 of the INTEL PL/M-86 Compiler User's Manual. BASIC follows the rules described for the MEDIUM case.

For illustrations of how the stack is altered after a call statement is given, in addition to the rules you must follow when coding a subroutine, see Appendix E of this manual.

CDBL Function

BRIEF

Format: `CDBL(X)`

Action: Converts X to a double-precision number.

Details

Many scientific, technical, and business applications require more digits than single-precision can provide. This is particularly true in programs where numeric quantities must be subjected to a long series of arithmetic processes.

Most operations performed on numeric data introduce small amounts of error. These errors tend to accumulate. At the end of a complex chain of operations, it is doubtful that you will have as many digits of precision as you started with. To ensure accurate results under these conditions, BASIC provides a double-precision type that uses eight bytes to represent real numbers to 16 decimal digits of accuracy (16 to 17 internally) instead of the seven digits (eight internally) attainable with the four byte, single-precision.

The CDBL function which converts numeric values to double-precision can help alleviate this problem of inaccuracy. If you convert the values to double-precision before the calculation is executed, you can then convert the values back to single precision (to save space) before printing or storing.

Example:

```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
 454.67 454.6700134277344
Ok
```

CHAIN Statement

BRIEF

Format: `CHAIN [MERGE] <filename>[, [<line number exp>] [,ALL] [,DELETE<range>]]`

Purpose: To call a program and pass variables to it from the current program.

Details

Filename

The <filename> in the CHAIN Statement is the name of the program that is called.

Example:

```
CHAIN"PROG1"
```

Line Number

<line number exp> is a line number or an expression that relates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line.

Example:

```
CHAIN"PROG1", 1000
```

<line number exp> is not affected by a RENUM command.

ALL option

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program **must** contain a COMMON statement to list the variables that are passed. The ALL option only works if a line number is specified. If a line number is not specified, no variables are passed if ALL is used. See Page 10.23.

Example:

```
CHAIN"PROG1", 1000,All
```

Overlay MERGE Option

If the MERGE option is included, it allows a subroutine to be brought into the BASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be merged.

Example:

```
CHAIN MERGE"OVRLAY",1000
```

After an overlay is brought in, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option.

Example:

```
CHAIN MERGE"OVRLAY2",1000,DELETE 1000-5000
```

The line numbers in <range> are not affected by the RENUM command.

The CHAIN statement with MERGE option leaves the files open and preserves the current OPTION BASE setting.

If the MERGE option is omitted, CHAIN does not preserve variable types or user defined functions for use by the chained program. Any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEF FN statements containing shared variables must be restated in the chained program.

The Microsoft BASIC compiler does not support the ALL, MERGE, DELETE, and <line number exp> options to CHAIN. Thus, the statement format is CHAIN <filename>. If you wish to maintain compatibility with the Microsoft BASIC compiler, it is recommended that COMMON be used to pass variables and that overlays not be used. The CHAIN statement leaves the files open during chaining.

When using the MERGE option, user defined functions should be placed before any CHAIN MERGE statements in the program. Otherwise, the user defined functions will be undefined after the merge is complete.

CHR$ Function

BRIEF

Format: `CHR$(I)`

Action: Returns a character which is the ASCII code for value I.

Details

The CHR$ function returns the character associated with the number enclosed in the parenthesis. It is the inverse function of the ASC function covered on Page 10.2.

CHR$ is commonly used to send a special character to the terminal. For instance, the bell character (CHR$ (7)) could be sent as a preface to an error message, or a form feed could be sent (CHR$(12)) to clear the terminal screen and return the cursor to the home position. (ASCII codes are listed in Appendix C.)

Example:

```
PRINT CHR$(66)
B
Ok
```

See the ASC function for ASCII-to-numeric conversion.

CINT Function

BRIEF

Format: `CINT(X)`

Action: Converts X to an integer by rounding the fractional portion. If X is not in the range -32768 to 32767, an "Overflow" error occurs.

Details

The CINT function converts X to an integer by rounding the fractional portion of the number to the closest whole number.

Example:

```
PRINT CINT(45.67)
 46
Ok
```

See the CDBL and CSNG functions for converting numbers to the double-precision and single-precision data types. See also FIX, Page 10.52 and INT, Page 10.80. Both return integers.

CIRCLE Statement

BRIEF

Format: `CIRCLE(Xcenter,Ycenter),radius`
`[,attribute[,start,end[,aspect]]]`

Purpose: To draw an ellipse with a center and radius as specified by the arguments.

Details

The CIRCLE statement draws an ellipse with a center and radius as indicated by the first of its arguments. The default attribute is the foreground color. The start and end angle parameters are radian arguments between 0 and 2* PI which allow you to specify where drawing of the ellipse will begin and end. If the start or end angle is negative, the ellipse will be connected to the center point with a line, and the angles will be treated as if they were positive (Note that this is different than adding 2*PI).

The aspect ratio describes the ratio of the X radius to the Y radius. The default aspect ratio is .4375 and will give a visual circle, assuming a standard monitor screen aspect ratio of 7/16.

If the aspect ratio is less than one, then the radius is given in X-pixels. If it is greater than one, the radius is given in Y-pixels. The standard relative notation may be used to specify the center point.

The start angle may be less than the end angle.

CLEAR Command

BRIEF

Format: `CLEAR [,[<expression1>][,< expression2>]]`

Purpose: To set all numeric variables to zero, all string variables to null, and to close all open files. Optionally, it sets the end of memory and the amount of stack space.

Details

<expression1> is a memory location which, if specified, sets the highest location available for use by BASIC.

<expression2> sets aside stack space for BASIC. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

The Microsoft BASIC compiler supports the CLEAR command with the restriction that <expression1> and <expression2> must be integer expressions. If a value of zero is given for either expression, the appropriate default is used. The default stack size is 256 bytes. The default top of memory is the current top of memory. The CLEAR command performs the following actions:

Closes all files
Clears all COMMON and user variables
Resets the stack and string space
Releases all disk buffers

Examples:

```
CLEAR
CLEAR ,32768
CLEAR ,,2000
CLEAR ,32768,2000
```

CLOSE Command

BRIEF

Format: `CLOSE[[#]<file number>[,[ # ]<file number...>]]`

Purpose: To conclude I/O to a disk file.

Details

The CLOSE command concludes I/O to a disk file. The <file number> is the number under which the file was opened. A CLOSE with no arguments closes all open files.

The relationship between a particular file and file number terminates upon execution of a CLOSE. The file may then be reopened using the same or different file number. Likewise, that file number may now be reused to open any file. A CLOSE for a sequential output file writes the final buffer of output.

The END statement and the NEW command always close all disk files automatically. (STOP does not close disk files.)

See Chapter 6, "File Handling", for more information concerning how the CLOSE command is used.

CLS Statement

BRIEF

Format: `CLS`

Purpose: The CLS statement erases the current screen.

Example:

```
1 CLS 'Clears the screen.
```

COLOR Statement

BRIEF

Format: `COLOR [Foreground] [, [Background]]`

Function: The COLOR statement selects the Foreground, and Background screen display colors.

Details

The COLOR statement is used to select the foreground colors and background colors for screen display. If you have a monochrome video board, this statement will be only partially effective. If you have a color video board but are using a monochrome monitor, your colors will appear in shades of gray. (The Z-100 All-in-One model has a green non-glare screen, thus your colors will appear in shades of green).

Foreground: = Foreground (for character color). An unsigned integer in the range zero to seven.

Background: = Background Color. An unsigned integer in the range of zero to seven.

Valid Colors are:

0 Black
1 Blue
2 Green
3 Cyan
4 Red
5 Magenta
6 Yellow
7 White

COLOR Statement

Rules:

1. Any values entered outside of the range 0-255 will result in an `Illegal Function Call` error. Previous values are retained.

2. Foreground color may equal background color. This has the effect of making any character displayed invisible. Changing the foreground or background color will make the characters visible again.

3. Any parameter may be omitted. Omitted parameters assume the old value.

4. The COLOR statement may not end in comma (,). For example COLOR 7 is legal and will leave the background unchanged.

Example:

`10 COLOR 7,0`	Select white forground, and black background.
`30 COLOR 6,4`	Change foreground to yellow, background to red.
`40 COLOR ,6`	Changes background to yellow, and any characters displayed on the screen.

COMMON Statement

BRIEF

Format: `COMMON <list of variables>`

Purpose: To pass variables to a chained program.

Details

The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program. It is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example:

```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
      .
      .
      .
```

COMMON Statement

Arrays in COMMON must be declared in preceding DIM statements.

The standard form of the COMMON statement is referred to as blank COMMON. FORTRAN style named COMMON areas are also supported; however, the variables are not preserved across chains. The syntax for named COMMON is as follows:

```
COMMON /<name>/ <list of variables>
```

where <name> is one to six alphanumeric character(s) starting with a letter. This is useful for communicating with FORTRAN and assembly language routines without having to explicitly pass parameters in the CALL statement.

The blank COMMON size and order of variables must be the same in the chaining and chained-to programs.

CONT Command

BRIEF

Format: `CONT`

Purpose: To continue program execution after a CTRL-C has been typed, or a STOP or END statement has been executed.

Details

When the CONT command is used, execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt or prompt string.

CONT is used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. CONT may be used to continue execution after an error.

CONT is invalid if the program has been edited during the break. Any modifications made to your program causes all variables to be set to zero.

See example provided on Page 10.161, for the STOP statement.

COS Function

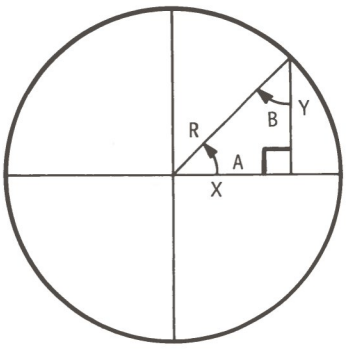
BRIEF

Format: `COS(X)`

Action: Returns the cosine of X in radians.

Details

The trigonometric (or circular) COS function, is best explained in relation to a circle (see figure below).



A given radius with length R defines a right triangle with base X, height Y and enclosed angles A and B. The ratios of the three sides of the triangle to one another can be expressed as functions of the angle A.

Specifically,

Y/R is SIN(A)
X/R is COS(A)
Y/X is TAN(A)

where SIN, COS, and TAN stand for sine, cosine, and tangent. These relationships can also be defined in terms of angle B.

The calculation of COS(X) is performed in single-precision.

Example:

```
10 X = 2*COS(.4)
20 PRINT X
RUN
 1.842122
Ok
```

To convert from degrees to radians, use the formula:

Radians = degrees * PI/180

where PI = 3.14159

CSNG Function

BRIEF

Format: `CSNG(X)`

Action: Converts X to a single-precision number.

Details

The CSNG function is used to convert a number to a single-precision number.

Example:

```
10 A#=975.321012345678
20 PRINT A#; CSNG(A#)
RUN
 975.3421012345678  975.3421
Ok
```

See the CINT and CDBL functions for converting numbers to integer and double-precision data types. Also see Chapter 5, "Converting Numeric Precisions", Page 5.51.

CSRLIN Function

BRIEF

Format: `X = CSRLIN`

Action: Returns the current line (or row) position of the cursor.

Details

The CSRLIN function returns the current Row position of the cursor. It is most often used with the POS function, which returns the column position.

x = CSRLIN x is a numeric variable receiving the value returned. The value returned will be in the range 1 to 25.

x = POS(0) will return the column location of the cursor. This value will be in the range 1 to 80.

Example:

```
10 Y = CSRLIN  'Record current line.
20 X = POS(I)  'Record current column.
30 LOCATE 24,1 :PRINT "HELLO" 'Print HELLO on the 24th line.
40 LOCATE Y,X  'Restore position to old line, column.
```

CVI, CVS, CVD Functions

BRIEF

Format: `CVI(<2-byte string>)`
`CVS(<4-byte string>)`
`CVD(<8-byte string>)`

Action: Converts string values to numeric values.

Details

Numeric values that are read from a random disk file must be converted from strings back into numbers. CVI converts a two-byte string to an integer. CVS converts a four-byte string to a single-precision number. CVD converts an eight-byte string to a double-precision number.

Example:

```
        .
        .
        .
70 FIELD #1,4 AS N$, 12 AS B$ ...
80 GET #1
90 Y=CVS(N$)
        .
        .
        .
```

See also MKI$, MKS$, MKD$, on Page 10.107, and Chapter 6, "File Handling", on Page 6.1.

DATA Statement

BRIEF

Format: `DATA <list of constants>`

Purpose: To store the numeric and string constants that are accessed by the program's READ statement(s). (See READ, Page 10.142)

Details

Data stored in DATA statements are constants that must be accessed sequentially. DATA statements are non-executable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas), and any number of DATA statements may be used in a program.

The READ statements access the DATA statements in order (by line number). The data contained in the data statements may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

The <list of constants> may contain numeric constants in any format, i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be re-read from the beginning by use of the RESTORE statement (Page 10.147).

See the examples in the discussion of the READ statement, Page 10.142.

DATE$ Statement

BRIEF

Format: `DATE$ = <string expr>` To set the current date.
`<string var> = DATE$` To get the current date.

Purpose: DATE$ statement may be used to set or retrieve the current date.

<string expr> Is a valid string literal or variable.

Details

The current date is returned and assigned to the string variable if DATE$ is the expression in a LET or PRINT statement.

The current date is stored if DATE$ is the target of a string assignment.

Rules:

1. If <string expr> is not a valid string, a `Type mismatch` error will result. Previous values are retained.

2. For <string var> = DATES$, DATE$ returns a 10 character string in the form `mm-dd-yyyy` where `mm` is the month (01 to 12), `dd` is the day (01 to 31), and `yyyy` is the year (1980 to 2077).

3. For DATE$ = <string expr>, <string expr> may be one of the following forms:

```
"mm-dd-yy"
"mm-dd/yy"
"mm-dd-yyyy"
or
"mm/dd/yyyy"
```

   If any of the values are out of range or missing, an `Illegal Function Call` error message is issued. Any previous date is retained.

Example:

```
DATE$ = "01-01-81"
Ok
PRINT DATE$
01-01-1981
Ok
```

DEF FN Statement

BRIEF

Format: `DEF FN<name>[(<parameter list>)]=`
`<function definition>`

Purpose: To define and name a function written by the user.

Details

Name

The <name> in a DEF FN function must be a legal variable name. This name, preceded by FN, becomes the name of the function. The <parameter list> is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas. The <function definition> is an expression that performs the operation of the function. It is limited to one line.

Variable names that appear in this expression serve only to define the function. They do not affect program variables that have the same name. A variable name used in a function may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

User-defined functions may be numeric or string. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name, and the argument type does not match, a `Type mismatch` error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an `Undefined user function` error occurs. DEF FN is illegal in the direct mode.

DEF FN Statement

Example:

```
          .
          .
410 DEF FNAB (X,Y)=X^3/Y^2
420 T=FNAB(I,J)
          .
          .
```

Line 410 defines the function FNAB. The function is called in line 420. An error in the function call will show up as an error in line 420 not in line 410 where it actually occurred. Therefore, you must look for the error in the line number in which the function was called.

DEFINT/SNG/DBL/STR Statements

BRIEF

Format: `DEF<type> <range(s) of letters>`
where <type> is INT, SNG, DBL, or STR

Purpose: To declare variable types as integer, single-precision, double-precision, or string.

Details

A DEF type statement declares that the variable names beginning with the letter(s) specified will be that type variable. All value assignments made to variables are cleared before a define type statement.

If no type declaration statements are encountered, BASIC assumes all variables without declaration characters are single-precision variables.

Examples:

`10 DEFDBL L-P`	All variables beginning with the letters L,M,N,O, and P will be double-precision variables.
`10 DEFSTR A`	All variables beginning with the letter A will be string variables.
`10 DEFINT I-N,W-Z`	All variables beginning with the letters I, J, K, L, M, N, W, X, Y, and Z will be integer variables.

DEF SEG Statement

BRIEF

Format: `DEF SEG [=<address>]`

Purpose: The DEF SEG statement assigns the current value to be used by a subsequent BLOAD, BSAVE, PEEK, POKE, CALL, or user defined function call.

Details

The <address> is a valid numeric expression returning an unsigned integer in the range 0 to 65535.

The address specified is saved for use as the segment required by the BLOAD, BSAVE, PEEK, POKE, and CALL statements.

Rules:

1. Any value entered outside of this range will result in an `Overflow` error message. The previous value is retained.

2. If the address option is omitted, the segment to be used is set to the BASIC data segment. This is the initial default value.

3. If the address option is given, it should be a value based upon a 16 byte boundary. For the BLOAD, BSAVE, PEEK, POKE, or CALL statements, the value is shifted left four bits to form the code segment address for the subsequent call instruction. BASIC does not perform additional checking to assure that the resultant segment + offset value is valid.

4. **NOTE**: DEF and SEG must be separated by a space! Otherwise, BASIC would interpret the statement, `DEFSEG=100` to mean: "assign the value 100 to the variable DEFSEG".

Example:

```
10 DEF SEG=&HFE00    'Set segment to Monitor ROM.
20 DEF SEG           'Restore segment to BASIC's DS
```

DEF USR Statement

BRIEF

Format: `DEF USR[<digit>]=<integer expression>`

Purpose: To specify the starting address of an assembly language subroutine.

Details

The <digit> may be any digit from zero to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine. See Appendix E, "BASIC Assembly Language Subroutines."

Any number of DEF USR statements may appear in a program to redefine the subroutine starting addresses, allowing access to as many subroutines as necessary.

Example:

```
        .
        .
        .
200 DEF USR0=24000
210 X=USR0 (Y^2/2.89)
        .
        .
        .
```

See also the CALL statement on Page 10.10.

DELETE Command

BRIEF

Format: `DELETE[<line number>][-<line number>]`

Purpose: To delete program lines.

Details

BASIC always returns to command level after a DELETE command is executed. If <line number> does not exist, an `Illegal Function Call` error message is displayed.

Examples:

`DELETE 40`	Deletes line 40
`DELETE 40-100`	Deletes lines 40 through 100, inclusive
`DELETE -40`	Deletes all lines up to and including line 40

DIM Statement

BRIEF

Format: `DIM <list of subscripted variables>`

Purpose: To specify the maximum values for array variable subscripts and allocates storage accordingly.

Details

The dimension statement is used to set up the maximum values for array variable subscripts and to allocate storage accordingly. If an array variable name is used without a DIM statement, the maximum value of its subscript is assumed to be 10. If a subscript is used that is greater than the maximum specified, a `Subscript out of range` error occurs. The minimum value for a subscript is always zero, unless otherwise specified with the OPTION BASE statement (see Page 10.121).

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

Example:

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
50 DATA 1,4,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41
    .
    .
    .
```

For additional information on the use of the DIM statement, see "Array Variables," on Page 5.12.

DRAW Statement

BRIEF

Format: `DRAW <string expression>`

Purpose: To combine the capabilities of other graphic statements into an object definition language.

Details

The DRAW statement combines most of the capabilities of the other graphics statements into an easy-to-use object definition language called Graphics Macro Language. The GML command is a single character within a string, optionally followed by one or more arguments.

MOVEMENT COMMANDS

Each of the following movement commands begin movement from the "current graphics position". This is usually the coordinate of the last graphics point plotted with another GML command, LINE, or PSET.

U[<n>]	Move up (scale factor * N) points
D[<n>]	Move down
L[<n>]	Move left
R[<n>]	Move right
E[<n>]	Move diagonally up and right
H[<n>]	Move diagonally up and left
G[<n>]	Move diagonally down and left
F[<n>]	Move diagonally down and right

The above commands move one unit if no argument is supplied.

M <X,Y> Move absolute or relative. If X is preceded by a "+" or "−", X and Y are added to the current graphics position, and connected with the current position with a line. Otherwise, a line is drawn to point X,Y from the current position.

PREFIX COMMANDS

The following prefix commands may precede any of the above movement commands:

B	Move but don't plot any points.
N	Move but return to original position when done.
A<n>	Set angle n. n may range from zero to three, where zero is zero degrees, one is 90, two is 180, and three is 270. Figures rotated 90 or 270 degrees are scaled so that they will appear the same size as with zero or 180 degrees on a monitor screen with the standard aspect ratio of 7/16.
C<n>	Set attribute n. n may range from zero to seven.
S<n>	Set scale factor. n may range from one to 255. The scale factor is multiplied by the distances given with U,D,L,R or relative M commands to get the actual distance traveled.
X<string;>	Execute substring (not supported by BASIC compiler). This powerful command allows you to execute a second substring from a string, much like GOSUB in BASIC. You can have one string execute another, which executes a third, and so on.

Numeric arguments can be constants like "123" or "=variable;", where variable is the name of a variable. (Not supported by BASIC compiler).

EDIT Command

BRIEF

Format: `EDIT <line number>`
`EDIT`

Purpose: To display the specified Line(s) and position the cursor under the first digit of the line number.

Details

The full screen editor recognizes special key combinations as well as numeric and cursor movement key-pad keys. These keys allow moving the cursor to a location on the screen, inserting characters, and deleting characters as described in chapter 3.

More than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last statement by a colon.

A Z-BASIC program line always begins with a line number, ends with a RETURN, and may contain a maximum of 250 characters.

With the full screen editor, the EDIT statement simply displays the line specified and positions the cursor under the first digit of the line number.

The format of the EDIT statement is:

```
EDIT <line number>     OR
EDIT
```

Line number is the program line number of a line existing in the program. If there is no such line, an `Undefined line number` error message is displayed.

A period (.) placed after the EDIT statement always gets the last line referenced by an EDIT statement, LIST command, or error message.

Remember, if you have just entered a line and wish to go back and edit it, the command **EDIT.** will enter EDIT at the current line. The line number symbol "." always refers to the current line.

END Statement

BRIEF

Format: `END`

Purpose: To terminate program execution, close all files, and return to command level.

Details

END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a `BREAK` message to be printed. An END statement at the end of a program is optional. BASIC always returns to command level after an END is executed.

Example:

```
520 IF K>1000 THEN END ELSE GOTO 20
```

EOF Function

BRIEF

Format: `EOF (<file number>)`

Purpose: Returns −1 (true) if the end of a sequential or random file has been reached.

Details

The EOF function is used to test for end-of-file while inputting, in order to avoid `Input past end` errors. If in a random access file, a GET is issued for a record that is past the end of the file, EOF will be set to −1, and no error will occur. A zero will be returned if the end of the file has not been reached. This function may be used to find the size of a file by using a binary search or other algorithm.

Example:

```
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
```

ERASE Statement

BRIEF

Format: `ERASE <list of array variables>`

Purpose: To eliminate arrays from a program.

Details

The ERASE statement can be used to make more storage space available while you are running your program by eliminating arrays from the program that are no longer needed.

Arrays may be redimensioned after they are erased, or, the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first erasing it, a `Duplicate Definition` error occurs.

The Microsoft BASIC compiler does not support ERASE.

Example:

```
        .
        .
        .
450 ERASE A,B
460 DIM B(99)
        .
        .
        .
```

ERR and ERL Variables

BRIEF

Format: `X = ERR`
`Y = ERL`

Purpose: To trap an error by returning an error code and line number associated with an error.

Details

When an error handling subroutine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct statement, use:

```
IF 65535 = ERL THEN ...
```

If the statement was an indirect mode statement use:

```
IF ERR = error code THEN ...

IF ERL = line number THEN ...
```

If the line number is not on the right side of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. The BASIC error codes are listed in Appendix A.

ERROR Statement

BRIEF

Format: `ERROR <integer expression>`

Purpose: 1) To simulate the occurrence of a BASIC error.

2) To allow error codes to be defined by the user.

Details

The value of <integer expression> must be greater than zero and less than 255. If the value of <integer expression> equals an error code already in use by BASIC (see Appendix A), the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed. (See example below.)

Example:

```
10 S = 10
20 T = 5
30 ERROR S + T
40 END
RUN
String too long in 30
Ok
```

Or, in direct mode:

```
Ok
ERROR 15                (you type this line)
String too long         (BASIC types this line)
Ok
```

ALPHABETICAL REFERENCE GUIDE

ERROR Statement

To define your own error code, use a value that is greater than any used by the Z-BASIC error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to Z-BASIC.) This user-defined error code may then be conveniently handled in an error trap routine.

Example:

```
      .
      .
      .
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B > 5000 THEN ERROR 210
      .
      .
      .
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL = 130 THEN RESUME 120
      .
      .
```

If an ERROR statement specifies a code for which no error message has been defined, BASIC responds with the message `Unprintable Error`. Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

EXP Function

BRIEF

Format: `EXP(X)`

Action: Calculates the exponential value of X.

Details

The EXP function returns the mathematical value of e to the power of X. X must be <=88.0296. If EXP overflows, the `Overflow` error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:

```
10 X = 5
20 PRINT EXP (X-1)
RUN
 54.59815
Ok
```

FIELD Statement

BRIEF

Format: `FIELD#<file number>,<field width> AS <string variable>`
`[,<field width> AS <string variable>...]`

Purpose: To allocate space for variables in a random file buffer.

Details

To get data out of a random buffer after a GET or to enter data before a PUT, a FIELD statement must have been executed.

The <file number> is the number under which the file was opened. <field width> is the number of characters to be allocated to <string variable>.

Example:

```
FIELD #1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does NOT place any data in the random file buffer. (See LSET/RSET and GET in chapter 6, "File Handling". Also refer to these statements in the Alphabetical Reference Guide.)

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was opened. Otherwise, a `FIELD overflow` error occurs. (The default record length is 128.)

Any number of FIELD statements may be executed for the same file, and all FIELD statements that have been executed are in effect at the same time.

Do not use a fielded variable name in an INPUT or LET statement. Once a variable name is fielded, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

FILES Command

BRIEF

Format: `FILES[<filename>]`

Purpose: To display the names of files residing on the current disk.

Details

If <filename> is omitted from a FILES command, all the files on the currently selected drive will be listed. <filename> is a string formula which may contain question marks (?) to match any character in the filename or extension. An asterisk (*) as the first character of the filename or extension will match any file or any extension.

Examples:

`FILES`	List all files on default drive
`FILES "*.BAS"`	List all files with .BAS extension
`FILES "B:*.*"`	List all files on disk B
`FILES "TEST?.BAS"`	List all files with a primary name that begins with "TEST" and has an an extension of .BAS. The question mark could be any alpha-numeric character.

FIX Function

BRIEF

Format: `FIX(X)`

Action: Returns the truncated integer part of X.

Details

The FIX function is used to truncate the integer portion of a number. FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for a negative X.

Examples:

```
PRINT FIX(58.75)
 58
Ok

PRINT FIX(-58.75)
-58
Ok
```

FOR...NEXT Statement

BRIEF

Format:
```
FOR <variable>=X TO Y [STEP z]
    .
    .
    .
NEXT [<variable>] [,<variable>...]
```

Purpose: To allow a series of instructions to be performed in a loop structure a given number of times.

The <variable> in the format of the FOR... NEXT statement is used as a counter.

X is the initial value of the counter and Y is the final value.

Details

For Next Loops

Looping is a common program structure used in programming applications when there is a need to repeat a series of instructions several times. The FOR...NEXT statements are used to keep track of how many times the program loops and to provide a way for the program to exit from the loop when the specified number of loops has been completed.

Counters

The <variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is increased by the amount specified by STEP. If no step value is specified, the default value is one.

Checks

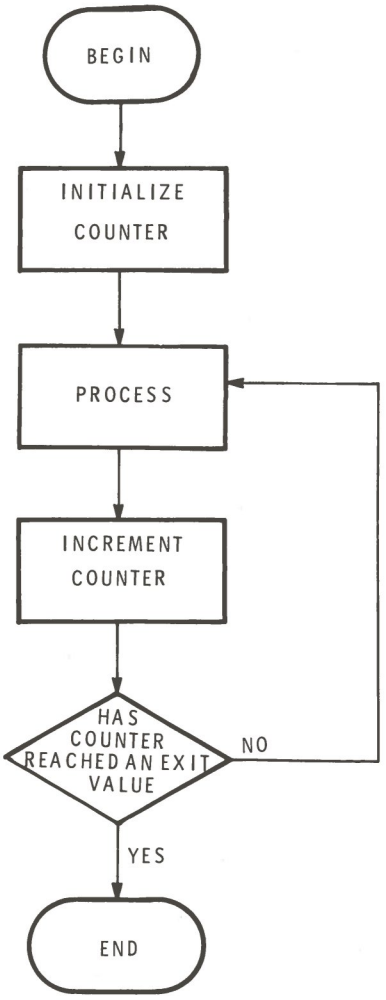
A check is then performed to see if the current value is below or equal to the final value. If it is, the process is repeated. If greater, execution continues with the statement following the NEXT statement.

We have included a flowchart of a FOR...NEXT loop to help you in understanding how these statements function. A flowchart is a graphic illustration, using standard symbols, to represent the path of a program.

ALPHABETICAL REFERENCE GUIDE

FOR...NEXT Statement

**FOR...
NEXT
FLOWCHART**



Now that you have seen the flow of the program, consider the examples below:

Example 1:

```
10 C=1
20 PRINT C
30 C=C+1
40 IF C<=10 THEN 20
```

is the same as

```
10 FOR C=1 TO 10
20 PRINT C
30 NEXT C
```

The first part of this example uses the IF...THEN statement to execute the loop. The second part of this example uses the FOR...NEXT sequence to execute the same loop with the same results. Notice the FOR...NEXT example is shorter and will run faster than the IF...THEN loop. If you compare the example to the flowchart above, you will understand the program structure of a counter-driven loop.

A run of this program will look like this:

```
RUN
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
Ok
```

Step

Notice, in the preceding example, STEP was not specified. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter must be set to less than the initial value.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step. For example:

```
20 FOR I=1 TO 0
30 PRINT I
40 NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value. It is also important to remember that the final value of the loop must be set before the initial value is set.

Nesting FOR...NEXT STATEMENTS

FOR...NEXT loops may be nested. That is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before the NEXT statement for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a `NEXT without FOR` error message is issued, and execution is terminated. If a FOR statement appears without a corresponding NEXT statement, a `FOR Without NEXT` error is issued, and execution is terminated.

ALPHABETICAL REFERENCE GUIDE

FOR...NEXT Statement

Following is an example of a nested FOR...NEXT statement that creates a multiplication table of the multiples of five thru nine.

Example 2

```
10 PRINT "    ";
20 FOR Z=5 TO 9
30 PRINT Z;" ";
40 NEXT Z
50 PRINT
60 PRINT "    ";STRING$(20,"-")
70 FOR X=5 TO 9
80 PRINT X;"|";
90 FOR Y=5 TO 9
100 PRINT X*Y;
110 NEXT Y
120 PRINT
130 NEXT X
140 END
```

```
 RUN
  5   6   7   8   9
 --------------------
5 | 25  30  35  40  45
6 | 30  36  42  48  54
7 | 35  42  49  56  63
8 | 40  48  56  64  72
9 | 45  54  63  72  81
```

Checkpoint

The first FOR...NEXT statement found in lines 20-40 prints the numbers 5-9 across the top of the table. The nested FOR...NEXT statement is found in lines 70-130. Within those line numbers is the process for computing the actual table. If you compare this program to the flowchart illustrated on Page 10.54, you will be able to see where the two loops begin and end.

Nesting is a fairly complicated program structure. If you are having problems understanding how to do this, refer to the bibliography of this manual for references to additional resources.

FRE Function

BRIEF

Format: `FRE(0)`
`FRE(X$)`

Action: Returns the number of bytes in memory not currently being used by BASIC.

Details

The FRE function will return the number of bytes in memory that are not being used by Z-BASIC. The arguments to FRE are dummy arguments. FRE(" ") forces some system housekeeping before returning the number of free bytes.

BE PATIENT: housekeeping may take as long as one and one half minutes. BASIC will not initiate housekeeping until all free memory has been used. Therefore, using FRE(" ") periodically will result in shorter delays for each housekeeping.

Example:

```
PRINT FRE(0)
 14542
Ok
```

GET Statement

BRIEF

Format: `GET #<file number>[,<record number>]`

Purpose: To read a record from a random disk file into a random buffer.

Details

The <file number> in a GET statement is the number under which the file was opened. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767. See Chapter 6, "File Handling".

After a GET statement, the variables are immediately accessable.

GET/PUT Statement

BRIEF

Format: `GET (X1,Y1)-(X2,Y2), array name`
Format: `PUT (X1,Y1) ,array[,action verb]`

Purpose: To transfer graphic images to and from the screen.

Details

The PUT and GET statements are used to transfer graphic images to and from the screen. PUT and GET make animation and high-speed object motion possible in either graphic mode.

The GET statement transfers the screen image bounded by the rectangle described by specified points into the array. The rectangle is defined the same way as the rectangle drawn by the LINE statement using the ",B" option.

The array is simply used as a place to hold the image and can be of any type except string. It must be dimensioned large enough to hold the entire image. The contents of the array after a GET will be meaningless when interpreted directly (unless the array is of type integer).

The PUT statement transfers the image stored in the array onto the screen. The specified point is the coordinate of the top left corner of the image. An `Illegal Function call` error will result if the image to be transferred is too large to fit on the screen.

The action verb is used to interact the transferred image with the image already on the screen. PSET transfers the data onto the screen verbatum. Other possible action verbs include: PRESET, AND, OR, XOR.

PRESET is the same as PSET except that a negative image (e.g. black on white) is produced.

AND is used when you want to transfer the image only if an image already exists under the transferred image.

OR is used to superimpose the image onto the existing image.

XOR is a special mode often used for animation. XOR causes the points on the screen to be inverted where a point exists in the array image. This behavior is exactly like the cursor on the screen. XOR has a unique property that makes it especially useful for animation: when an image is put against a complex background once twice, the background is restored unchanged. This allows you to move an object around the screen without obliterating the background.

The default action mode is XOR.

It is possible to get an image in one mode and put it in another, although the effect may be quite strange because of the way points are represented in each mode.

ANIMATION

Animation of an object is usually performed as outlined below:

1. PUT the object(s) on the screen.

2. Recalculate the new position of the object(s).

3. PUT the object(s) on the screen a second time at the old location(s) to remove the old image(s).

4. Go to step one, this time putting the object(s) at the new location.

Movement done this way will leave the background unchanged. Flicker can be cut down by minimizing the time between steps four and one, and by making sure that there is enough time delay between one and three. If more than one object is being animated, every object should be processed at once, one step at a time.

If it is not important to preserve the background, animation can be performed using the PSET action verb. The idea is to leave a border around the image when it is first gotten as large or larger than the maximum distance the object will move. Thus, when an object is moved, this border will effectively erase any points.

The storage format in the array is as follows:

2 bytes giving X dimension in bits
2 bytes giving Y dimension
The array data itself

The data for each row of pixels is left justified on a byte boundary, so if there are less than a multiple of eight bits stored, the rest of the byte will be filled out with zeros. The required array size in bytes is:

4 + INT((X + 7)/8)*3*Y

WHERE: bits per pixel is 3

X = number of colums to be stored

Y = number of rows to be stored

The bytes per element of an array are:

2 for integer %
4 for single-precision !
8 for double-precision #

Example:

If you wanted to GET a 10 by 12 image into an integer array the number of bytes required is 4 + INT((10 + 7)/8)3*12 or 76 bytes. You would then divide the number of bytes by the number of bytes per element. In this case, 76/2. Thus, you would need an integer with at least 38 elements. See pages 8.20-8.22 for further information regarding the calculation of the array size.

It is possible to examine the X and Y dimensions and even the data itesIf if an integer array is used. The X dimension is in element zero of the array, and the Y dimension is found in element one. It must be remembered, however, that integers are stored low byte first, then high byte, but the data is transferred high byte first (leftmost).

GOSUB...RETURN Statement

BRIEF

Format:

```
GOSUB<line number>
  .
  .
  .
RETURN
```

Purpose: To branch to and return from a subroutine.

Details

The <line number> in the format of the GOSUB...RETURN statement is the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program. It is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement to direct program control around the subroutine.

Example:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

GOTO Statement

BRIEF

Format: `GOTO <line number>`

Purpose: To branch unconditionally out of the normal program sequence to a specified line number.

Details

The GOTO statement forces the program to branch unconditionally to the specified line number and continue execution of the program from that point. If the line number is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after the line number.

USED WITH IF...THEN

Although the GOTO statement is not a decision making statement, it is often used in conjunction with them. Alone, the GOTO statement will only cause the program to branch to another segment of the program. But, when used with a decision maker such as the IF...THEN statement, it becomes the object of a conditional branch, that is executed **only** if the result of the condition is true.

Checkpoint

To check your understanding of the GOTO statement, consider the following program:

```
10 print "LINE 10 HERE"
20 GOTO 40
30 PRINT "LINE 30 HERE"
40 PRINT "LINE 40 HERE"
```

Your understanding of the GOTO statement is clear if you imagined that a run of this program would look like this:

```
RUN
LINE 10 HERE
LINE 40 HERE
Ok
```

Line 30 has become inoperative and was totally ignored by the program because of the unconditional program branch instruction in line 20.

HEX$ Function

BRIEF

Format: `HEX$(X)`

Action: Returns a string which represents the hexadecimal value of the decimal argument evaluated.

Details

The HEX$ function returns the hexadecimal value of a decimal argument. X is rounded to an integer before HEX$ (X) is evaluated.

Example:

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL"
RUN
? 32
 32  DECIMAL IS 20 HEXADECIMAL
Ok
```

See the OCT$ Function for octal conversion.

IF Statement

BRIEF

Format: `IF <expression> THEN <statement(s)>|<line number>`
`[ELSE <statement(s)>|<line number>]`

`IF <expression> GOTO <line number>`
`[ELSE <statement(s)>|<line number>]`

Purpose: To make a decision regarding program flow based on the result returned by an expression.

Details

Conditional branching allows a program to take one or more program paths, depending on the result of an expression. The IF...THEN statement is one way to maintain program control when an expression is evaluated as true. If the result of an expression is true, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching, or one or more statements to be executed. If the result of the expression is false, the THEN clause is ignored and the ELSE clause if present is executed. Execution continues with the next executable statement.

GOTO

GOTO statements are always followed by a line number. If the result of the expression is false, the GOTO clause is ignored and the ELSE clause, if present, is executed. Execution proceeds at the first executable statement encountered after the line number.

ALPHABETICAL REFERENCE GUIDE

IF Statement

Example 1:

For an example of how these statements interact, input the example below:

```
10 REM  *********QUADRATIC ROOTS *********
20 REM
30 REM FIND THE TWO ROOTS, X1 AND X2, OF A QUADRATIC
40 REM EQUATION GIVEN COEFFICIENTS A,B,C
50 REM
100 REM INITIALIZE A,B,AND C,
110 PRINT: INPUT "COEFFICIENTS (A,B,C): ";A,B,C
200 REM CALCULATE ROOTS
210 X1=(-B+SQR(B^2-4*A*C))/(2*A)
220 X2=(-B-SQR(B^2-4*A*C))/(2*A)
300 REM PRINT OUT RESULTS
310 PRINT: PRINT "X1 IS ";X1:PRINT "X2 IS ";X2:PRINT
320 END
```

If you run this program a few times inserting random numbers for the coefficients (both negative and positive), you will notice that on many occasions the program ends with the following error message:

```
Illegal Function Call in 210
```

This happens when the values you enter for A, B, and C can not be calculated in the real number system. There are no real number values that equate to the square root of a negative number. Thus if you enter coefficients of 1,0,1, an error message would be displayed and the program terminated.

To maintain program control no matter what the values of A, B, and C are, and to keep certain values away from the square root formula, we have inserted a data check which cause the program to have two paths to choose. IF the value of B ^ 2 – 4*A*C is less than zero THEN the program will branch to an error message printing routine.

Example 2:

```
10 REM  *********QUADRATIC ROOTS *********
20 REM
30 REM FIND THE TWO ROOTS, X1 AND X2, OF A QUADRATIC
40 REM EQUATION GIVEN COEFFICIENTS A,B,C
50 REM
100 REM INITIALIZE A,B,AND C,
110 PRINT: INPUT "COEFFICIENTS (A,B,C): ";A,B,C
150 REM CHECK DATA
160 IF (B^2)-(4*A*C) <0 THEN 400
200 REM CALCULATE ROOTS
210 X1=(-B+SQR(B^2-4*A*C))/(2*A)
220 X2=(-B-SQR(B^2-4*A*C))/(2*A)
300 REM PRINT OUT RESULTS
310 PRINT: PRINT "X1 IS ";X1:PRINT "X2 IS ";X2:PRINT
320 GOTO 999
400 REM PRINT MESSAGE
410 PRINT: PRINT "NO REAL ROOTS.":PRINT
999 END
```

Line 160 contains the data check. If the value of B ^ 2–4*A*C is less than zero then the result is said to be true, and the program branches to line 400 printing the message, "NO REAL ROOTS". If the value is greater than zero, then the program continues execution at line 200. Using conditional branching keeps the program under your control, no matter what values are input.

Nesting

IF...THEN... ELSE statements can be nested. The term nesting means to embed a statement or any block of statements within a larger statement or block of statements. Nesting is limited only by the maximum length of the line, 255 characters. The ELSE must be in the same program line as the IF...THEN clause. In the example below, the statement may appear to be on two lines but it is still considered one program line if there is no intervening carriage return.

Example 3:

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

is a legal statement which means if the value of Y is greater than X, then the first part of this statement is true, and the rest of this statement is ignored. `GREATER` will be printed and the program will continue execution at the next line. If Y is less than X, then print, `LESS THAN` will be printed. If both of these statements are false, `"EQUAL"` will be printed.

ALPHABETICAL REFERENCE GUIDE

IF Statement

If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN.

Example 4:

```
IF A=B THEN IF B=C THEN PRINT "A=C"
     ELSE PRINT "A<>C"
```

BASIC will look at the first part of this statement (IF A=B). If it is false and because there is no corresponding ELSE statement, it will move to the next program line without printing anything. If the first part of the statement is true, but the second part (IF B=C) is false, then, it will print `A<>C` because the ELSE clause of the statement is matched to the closest unmatched THEN. If both parts of the statement are true then BASIC will print `A=C`.

Checkpoint

To test your understanding of nested IF...THEN statements, study the example below and match the ELSE clauses to the correct IF...THEN statements.

```
10 INPUT A: INPUT B: INPUT C
20 IF A=C THEN IF A=B THEN PRINT "A=B A=C"
<operator-typed LINE FEED>
     ELSE PRINT "A NOT = B"
<operator-typed LINE FEED>
     ELSE PRINT "A NOT =C"
30 PRINT A, B, C
```

This nested IF will first test to see if A=C. If A does not equal C, the second ELSE will be executed. If A does not equal B, the message `A NOT=C` will be printed and execution will be continued in line 30.

If A=C, the first THEN will be executed. This will result in another test. This time, A will be compared to B. If A does not equal C, the message `A NOT =B` will be printed, and execution will be continued in line 30.

If you understand how these statements were matched, you may want to read the next page for the additional technical considerations. If you're still a little unclear, don't worry. Nesting is a fairly complex program structure that may require additional reading. Resources may be found in the bibliography of this manual.

TECHNICAL DATA

If an IF...THEN statement is followed by a line number in the direct mode, an `Undefined line number` error results unless a statement with the specified line number was previously entered in the indirect mode.

When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0 use:

```
IF ABS(A-1.0)<1.0E-6 THEN...
```

This test is true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Following are three additional examples of using IF...THEN statements:

```
200 IF I THEN GET#1,I
```

This statement gets record number I if I is not zero.

```
100 IF(I<20)AND(I>10) THEN DB=1979-I:GOTO 300
110 PRINT "OUT OF RANGE"
      .
      .
      .
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated, and execution branches to line 300. If I is not in this range, execution continues with line 110.

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This statement causes printed output to go either to the terminal or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer, otherwise output goes to the terminal.

Complex conditions are explained in Chapter 5, "Logical Operators," Page 5.32.

INKEY$ Variable

BRIEF

Format: `X$=INKEY$`

Purpose: To read a character from the keyboard.

Details

The returned value is a zero, one, or two, character string.

A null string, (zero length) indicates no character is pending at the keyboard.

A one character string will contain the actual character read from the keyboard.

A two character string indicates a special extended code.

If the INKEY$ variable is in use, no characters are displayed on the screen and all characters are passed through to the program except for Control-C which terminates the program.

You must assign the result of INKEY$ to a string variable before using the character with any BASIC statement or function

Example:

```
100 'stop program until a key is pressed
110 PRINT "PRESS ANY KEY TO CONTINUE"
120 A$=INKEY$: IF A$="" THEN 120
```

Also see INPUT$ function, Page 10.78.

INP Function

BRIEF

Format: `INP(I)`

Purpose: Returns the byte read from port I.

Details

I must be in the range −32768 to 32767. The INP function is the complementary function to the OUT statement, Page 10.122.

Example:

```
100 A=INP(255)
```

INPUT Statement

BRIEF

Format: `INPUT [;] [<"prompt string">;] <variable list>`

Purpose: To allow input from the keyboard during program execution.

Details

Input Statements

Most programs have the following capabilities: *get data, process data,* and *print results*. The INPUT statement is one method of getting data from the keyboard. When an INPUT statement is encountered, program execution stops, a prompt string is printed if one has been included, a question mark is displayed (unless suppressed by a comma) and BASIC waits for your input of data. After receiving the proper response, program execution continues.

Prompt Strings

A prompt string can be included in an INPUT statement to remind you of the value the input statement is requesting. This is particularly useful when your programs use many input statements. Additionally, a prompt string advises you as to what type of response is appropriate. Following on the next page is an example of the use of a prompt string. Program 1 is a sample program using the INPUT statement without a prompt string. Program 2 is a modification of Program 1 with the prompt string included.

```
10 REM     *******PYTHAGOREAN THEOREM**********
20 REM
30 REM GIVEN TWO SIDES A AND B OF A RIGHT TRIANGLE,
40 REM FIND THE HYPOTENUSE, C
50 REM
100 INPUT A
110 INPUT B
120 C=SQR(A^2+B^2)
130 PRINT "THE HYPOTENUSE IS";C
140 END
```

Program 1

INPUT Statement Without Prompt String

When this Program is run it will look like this:

```
RUN
? 79
? 53
THE HYPOTENUSE IS 95.13149
Ok
```

```
10 REM    *******PYTHAGOREAN THEOREM*******
20 REM
30 REM GIVEN TWO SIDES A AND B OF A  RIGHT TRIANGLE,
40 REM FIND THE HYPOTENUSE, C
50 REM
100 INPUT "LENGTH OF SIDE A";A
110 INPUT "LENGTH OF SIDE B";B
120 C=SQR(A^2+B^2)
130 PRINT "THE HYPOTENUSE IS";C
140 END
```

Program 2.

INPUT Statement With Prompt String

When the Program is run, it will look like this:

```
RUN
LENGTH OF SIDE A?  79
LENGTH OF SIDE B?  53
THE HYPOTENUSE IS  95.13149
Ok
```

ALPHABETICAL REFERENCE GUIDE

INPUT Statement

Semicolons and Commas

You will notice that in the format of an INPUT statement, there is an optional semicolon included immediately following INPUT. In this case, the carriage return typed by the user to input data does not echo a carriage return/line feed sequence. This means that the cursor will remain on the same line as the user's response. A comma may be used instead of a semicolon after the prompt string to suppress the question mark.

Example:

10 INPUT "ENTER YOUR NAME",N$

will run as

```
ENTER YOUR NAME_
```

Variable List

Data entered are assigned to the variable(s) given in the variable list. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is inputted must agree with the type specified by the variable name. (Strings entered in response to an input statement need not be surrounded by quotation marks.)

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

Checkpoint

As a final example of using INPUT statements, we have included another sample program. This application program calculates mortgage payments. Enter the program and then study it to see how it works. Then, run it a few times and take note of the results. Be sure to save the program as you may want to modify it later.

```
10 REM  ***********MORTGAGE PAYMENTS*********
20 REM
30 REM CALCULATE MONTHLY MORTGAGE PAYMENTS GIVEN THE
40 REM TOTAL COST, DOWN PAYMENT, NUMBER OF YEARS,
50 REM AND YEARLY INTEREST RATE
60 REM
100 REM GET DATA
110 PRINT
120 PRINT "ENTER THE FOLLOWING:"
130 INPUT "TOTAL COST OF HOUSE AND PROPERTY:   ",T
140 INPUT "DOWN PAYMENT: ",D
150 INPUT "NUMBER OF YEARS FOR LOAN:   ",NY
160 INPUT "YEARLY INTEREST RATE (E.G. 16).   ",IY
170 PRINT
180 REM CALCULATE PRINCIPAL
200 P=T-D
220 REM CALCULATE MONTHLY RATE & CHANGE % TO DECIMAL
230 IM=IY/1200
240 REM CALCULATE TOTAL NUMBER OF PAYMENTS
250 NM=NY*12
300 REM CALCULATE PAYMENTS ETC. & REPORT RESULTS
310 PRINT
320 PRINT "YOUR PRINCIPAL IS $";P
330 MP=(P*IM*(1+IM)^NM)/((1+IM)^NM-1)
340 PRINT "THE MONTHLY PAYMENT WILL BE $";MP
350 PRINT "THE TOTAL PAYMENT FOR ";NY;" YEARS WILL BE $";NM*MP
360 PRINT "THE TOTAL INTEREST PAID WILL BE $";NM*MP-P
999 END
```

ALPHABETICAL REFERENCE GUIDE

INPUT Statement

When the mortgage payments program is run it should look like this:

```
RUN

ENTER THE FOLLOWING:
TOTAL COST OF HOUSE AND PROPERTY:     120000
DOWN PAYMENT:      40000
NUMBER OF YEARS FOR LOAN:     30
YEARLY INTEREST RATE (E.G., 16):      16

YOUR PRINCIPAL IS $ 80000
THE MONTHLY PAYMENT WILL BE $ 1075.806
THE TOTAL PAYMENT FOR  30 YEARS WILL BE $ 387290.1
THE TOTAL INTEREST PAID WILL BE $ 307290.1
```

The formula used to calculate the mortgage program was:

$$A = \frac{Pi(1+i)^n}{(1+i)^n - 1}$$

which translates into the BASIC assignment statement:

```
MP=(P*IM*(1+IM) ^ NM)/((1+IM) ^ NM-1)
```

where:
- MP= monthly payment
- P = principal
- IM= monthly interest rate
- NM= number of monthly payments

INPUT# Statement

BRIEF

Format: `INPUT#<file number>,<variable list>`

Purpose: To read data items from a sequential disk file and assign them to program variables.

Details

The INPUT# statement is used to read data items from a sequential disk file and assign them to program variables. The <file number> is the number used when the file was opened for input. The <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed, as with INPUT.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. Numeric values, leading spaces, carriage returns and line feeds are ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed, or comma.

If BASIC is scanning the sequential data file for a string item, leading spaces, carriage returns, and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second.

Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string and will terminate when it reaches a comma, return, or line feed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

See Chapter 6, "File Handling," Page 6.1.

INPUT$ Function

BRIEF

Format: `INPUT$(X,[[#]Y])`

Action: Returns a string of X characters, read from the terminal or from file number Y.

Details

If the terminal is used for input, no characters will be echoed, and all control characters are passed through except CTRL-C, which is used to interrupt the execution of the INPUT$ function.

Example 1:

```
5  'LIST THE CONTENTS OF A SEQUENTIAL FILE IN
    HEXADECIMAL
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT: CLOSE
60 END
```

Example 1 opens a disk file called DATA (line 10). It then reads one character at a time until the end of file (EOF) is reached (line 20). As each character is read, it is converted into ASCII value and then into its hexadecimal value and printed as such. The input and conversion is done in line 30.

Example 2:

```
      .
      .
      .
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
      .
      .
      .
```

INSTR Function

BRIEF

Format: `INSTR([I,]X$,Y$)`

Action: Searches for the first occurrence of the string Y$ in X$ and returns the position at which the match is found.

Details

Optional offset I sets the position for starting the search. I must be in the range one to 255. If I>LEN(X$), if X$ is null or if Y$ cannot be found, the INSTR function returns one. If Y$ is null, INSTR returns I or one. X$ and Y$ may be string variables, string expressions or string literals.

Example:

```
10 X$ = "ABCEDB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2  6
Ok
```

IF I<=0 or I>255 is specified, the error message `Illegal Function Call in <line number>` will be returned.

INT Function

BRIEF

Format: `INT(X)`

Action: The INT function returns the largest integer less than X.

Details

Examples:

```
PRINT INT(99.89)
 99
Ok

PRINT INT(-12.11)
-13
Ok
```

See FIX, Page 10.52, and CINT, Page 10.16, which also return integer values.

KEY Statement

BRIEF

Format: `KEY <key number>,<string expression>`
`KEY LIST`
`KEY ON`
`KEY OFF`

Purpose: To allow any of the twelve special function keys to be assigned to a 15 byte string which, when the key is pressed, will be input to BASIC.

Details

The KEY statement allows function keys to be designated "Soft Keys". Any one or all of the twelve special function keys may be assigned a 15 byte string which, when the key is depressed, will be inputted to BASIC.

Initially, the Soft Keys are assigned the following values:

F1—LIST	F7 —AUTO
F2—RUN	F8 —FOR
F3—LOAD"	F9 —NEXT
F4—SAVE"	F10—GOSUB
F5—CONT	F11—TRON
F6—PRINT	F12—TROFF

NOTE: F2, F5, F11 and F12 are executed immediately, because a carriage return is appended at the end.

<**key number**> is the key number. An expression returning an unsigned integer in the range one to 12.

<**string expression**> is the key assignment test, which can be any valid string expression.

KEY ON Causes the key values to be displayed on the 25th Line. 10 of the 12 soft keys are displayed. Only the first six characters of each value are displayed.

ALPHABETICAL REFERENCE GUIDE

KEY Statement

KEY OFF Erases the Soft Key display from the 25th line.

KEY LIST Lists all 12 Soft Key values on the screen. All 15 characters of each value are displayed.

KEY <key number>,<string expression> Assigns the string expression to the Soft Key specified (1 to 12).

Rules:

1. If the value returned for <key number> is not in the range one to 12, an `Illegal Function Call` error is taken. The previous key string assignment is retained.

2. The key assignment string may be one to 15 characters in length. If the string is longer than 15 characters, the first 15 characters are assigned.

3. Assigning a null string (string of length zero) to a Soft Key disables the function key as a Soft Key.

4. When a Soft Key is assigned, the INKEY$ function returns one character of the Soft Key string per invocation. If the Soft Key is disabled, INKEY$ returns a string of length two. The first character is binary zero, the second is the key scan Code.

Examples:

`50 KEY ON`	Display the Soft Key on the 25th Line.
`200 KEY OFF`	Erase Soft Key display
`10 KEY 1,"MENU"+CHR$(13)`	Assigns the string 'MENU'<carriage return> to Soft Key 1. Such assignments might be used for rapid data entry. This example might be used in a program to select a menu display when entered by the user.
`20 KEY 1,""`	Would erase Soft Key 1.

The following routine initializes the first five Soft Keys:

```
1 KEY OFF 'Turn off key display during init.
10 DATA KEY1,KEY2,KEY3,KEY4,KEY5
20 FOR I=1 TO 5:READ SOFTKEYS$(I)
30 KEY I,SOFTKEYS$(I)
40 NEXT I
50 KEY ON 'now display new softkeys.
```

Following is a practical application of the KEY statement you can use to RUN the DEMO programs on Pages 8.27-8.30. Input this program before you run the Demo.

```
10 KEY OFF
20 KEY 1,"RUN"
30 KEY 2,CHR$(34)+"DEMOI"+CHR$(34)+CHR$(13)
40 KEY 3,CHR$(34)+"DEMOII"+CHR$(34)+CHR$(13)
50 KEY 4,"LIST"+CHR$(13)
60 KEY ON
```

KILL Command

BRIEF

Format: `KILL<filename>`

Purpose: To delete a file from disk.

Details

KILL is used for all types of disk files: program files, random data files, and sequential data files.

If a KILL Command is given for a file that is currently open, a `File already open` error occurs.

Example:

```
200 KILL "FILE.BAS"
```

See also Chapter 6, "File Handling" Page 6.1.

Note: Kill does not assume .BAS extension.

LEFT$ Function

BRIEF

Format: `LEFT$(X$,I)`

Action: Returns a string comprised of the leftmost I characters of X$.

Details

The LEFT$ function forms a substring from the left end of a source string. In reference to the format, I must be in the range zero to 255. If I is greater than LEN(X$), the entire string (X$) will be returned. If I=O, the null string (length zero) is returned.

Example:

```
10 A$ = "BASIC"
20 B$ = LEFT$(A$,3)
30 PRINT B$
RUN
BAS
Ok
```

Also see "MID$" Page 10.105 and "RIGHT$," Page 10.150.

LEN Function

BRIEF

Format: `LEN(X$)`

Action: Returns the number of characters in X$. Non-printing characters and blanks are counted.

Details

The LEN function returns the length of X$ in characters.

Example:

```
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
RUN
 16
Ok
```

LET Statement

BRIEF

Format: `[LET] <variable>=<expression>`

Purpose: To assign the value of an expression to a variable.

Details

The LET statement is optional, i.e., the equal sign is sufficient when assigning an expression to a variable name.

Example:

```
110 LET D=12
120 LET E=12 ^ 2
130 LET F=12 ^ 4
140 LET SUM=D+E+F
  .
  .
  .
```

is equivalent to

```
110 D=12
120 E=12 ^ 2
130 F=12 ^ 4
140 SUM=D+E+F
  .
  .
  .
```

ALPHABETICAL REFERENCE GUIDE

LINE Statement

BRIEF

Format: `LINE [(X1,Y1)]-(X2,Y2) [,[attribute]] [,b[f]]`

Purpose: To permit the drawing of lines in absolute and relative locations on the screen.

Details

LINE is the most powerful of the graphics statements. It allows a group of pixels to be controlled with a single statement. A pixel is the smallest point that can be plotted on the screen.

The simplest form of line is:

```
LINE -(X2,Y2)
```

This will draw from the last point to the point (X2,Y2) in the foreground attribute.

We can include a starting point also:

```
LINE(0,0)-(639,224) 'draw diagonal line down screen
LINE (0,100)-(639,100) 'draw bar across screen
```

We can append a color argument to draw the line in green, which is color two:

```
LINE (10,10)-(20,20),2 'draw in color 2!

10 CLS
20 LINE -(RND*639,RND*224),RND*7
30 GOTO 20
```

(Draws lines forever using random attribute.)

The final argument to line is ",b" -- box or ",bf" — filled box. The syntax indicates that we can leave out the attribute argument and include the final argument as follows:

```
LINE (0,0)-(100,100),,b  'draw box in foreground attribute.

LINE (0,0)-(200,200),2,bf 'filled box attribute 2
```

The ",b" tells BASIC to draw a rectangle with the points (X1,Y1) and (X2,Y2) as opposite corners. This avoids giving the four LINE commands:

```
LINE  (X1,Y1)-(X2,Y2)
LINE  (X1,Y1)-(X1,Y2)
LINE  (X2,Y1)-(X2,Y2)
LINE  (X1,Y2)-(X2,Y2)
```

which perform the equivalent function.

The ",bf" means draw the same rectangle as ",b" but also fill in the interior points with the selected attribute.

When out of range coordinates are given in the line command, the coordinate which is out of range is given the closest legal value. In other words, negative values become zero, Y values greater than 224 become 224 and X values greater than 639 become 639.

In the examples and syntax the coordinate form STEP (Xoffset, Yoffset) is not shown. However, this form can be used wherever a coordinate is used. Note that all of the graphic statements and functions update the last point referenced. In a line statement if the relative form is used on the second coordinate it is relative to the first coordinate.

Example:

```
10 CLS
20 LINE-(RND*639,RND*224),RND*7,bf
30 GO TO 20
```

In this example, the LINE statement is used to draw filled boxes at random locations on the screen. Since the color argument is also randomized, these boxes will appear in various shades or colors. This example is also a continuous loop. You will have to press **CTRL–C** to break program execution. For more information on this statement, see Chapter 8, "Advanced Color Graphics".

LINE INPUT Statement

BRIEF

Format: `LINE INPUT [;][<"prompt string">;]<string variable>`

Purpose: To input an entire line (up to 255 characters) to a string variable, without the use of delimiters.

Details

The prompt string is a string literal printed at the terminal before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the RETURN is assigned to <string variable>. If a line feed/RETURN sequence (this order only) is encountered, both characters are echoed. The RETURN is ignored. The line feed is put into <string variable>, and data input continues.

If the LINE INPUT statement is immediately followed by a semicolon, the RETURN you type to end the input line does not echo a RETURN/line feed sequence at the terminal.

A LINE INPUT may be aborted by typing **CTRL-C**. BASIC will return to command level and display **OK**. Typing **CONT** resumes execution at the LINE INPUT.

See example, Page 10.91, LINE INPUT#.

LINE INPUT# Statement

BRIEF

Format: `LINE INPUT#<file number>,<string variable>`

Purpose: To read an entire line (up to 255 characters), without delimiters, from a sequential disk data file to a string variable.

Details

A <file number> is the number under which the file was opened. A <string variable> is the variable name that the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a RETURN. Then it skips over the RETURN/line feed sequence, and the next LINE INPUT# reads all characters up to the next RETURN. (If a line feed/RETURN sequence is encountered, it is preserved.)

The LINE INPUT# statement is especially useful if each line of a data file has been broken into fields, or if a BASIC program saved in ASCII mode is being read as data by another program.

Example:

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1,C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1,C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION?  LINDAJONES   234,4  MEMPHIS
LINDA JONES   234,4   MEMPHIS
Ok
```

ALPHABETICAL REFERENCE GUIDE

LIST Command

BRIEF

Format 1: `LIST [<line number>]`

Format 2: `LIST [<line number>[-[<line number>]]]`

Purpose: To list all or part of the program currently in memory at the terminal.

Details

BASIC always returns to command level after a LIST command is executed.

Format 1: If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either by the end of the program or by typing **CTRL-C**.) If <line number> is included, only the specified line will be listed.

Format 2: This format allows the following options:

1. If only the first number is specified, that line and all higher-numbered lines are listed.

2. If only the second number is specified, all lines from the beginning of the program through that line are listed.

3. If both numbers are specified, the entire range is listed.

Examples:

Format 1:

`LIST`	Lists the program currently in memory.
`LIST 500`	Lists line 500.

Format 2:

`LIST 150-`	Lists all lines from 150 to the end.
`LIST-1000`	Lists all lines from the lowest number through 1000.
`LIST 150-1000`	Lists lines 150 through 1000, inclusive.

ALPHABETICAL REFERENCE GUIDE

LLIST Command

BRIEF

Format: `LLIST [<line number>[-[<line number>]]]`

Purpose: To list all or part of the program currently in memory at the line printer.

Details

The LLIST command is used to list all or part of a program at the line printer. LLIST assumes a 255-character wide printer.

BASIC always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST.

See the examples for LIST, Page 10.93.

LOAD Command

BRIEF

Format: `LOAD <filename>[,R]`

Purpose: To load a file from disk into memory.

Details

The <filename> in the LOAD command is the name that was used when the file was saved. The operating system appends a default filename extension of .BAS if one was not supplied in the SAVE command. (Refer to Chapter 2, "Files and File Naming" Page 2.12, for information about possible filename extensions under Z-DOS Operating System.)

R option

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program.

However, if the "R" option is used with LOAD, the program is run after it is loaded and all open data files are kept open. Thus, LOAD with the "R" option may be used to chain several programs (or segments of the same program.) Information may be passed between the programs using their disk data files.

Example:

```
LOAD "STRTRK",R
```

LOC Function

BRIEF

Format: `LOC(<file number>)`

Action: With random disk files, LOC returns the record number just read or written from a GET or PUT statement.

Details

If the file was opened but no disk I/O has been performed yet, the LOC function returns a zero. With sequential files, LOC returns the number of sectors (128 byte blocks) read from or written to the file since it was opened.

Example:

```
200 IF LOC(1)>50 THEN STOP
```

LOCATE Statement

BRIEF

Format: `LOCATE[row],[col][,[cursor]]`

Purpose: The LOCATE statement moves the cursor to the specified position on the active screen. Optional parameters turn the blinking cursor on and off.

Details

row Is the screen line number. A numeric expression returning an unsigned integer in the range 1 to 25.

col Is the screen column number. A numeric expression returning an unsigned integer in the range 1 to 40.

cursor Is a Boolean value indicating whether the cursor is visible or not: Zero for off, non-zero for on.

The LOCATE Statement moves the cursor to the specified position. Subsequent PRINT statements begin placing characters at this location. Optionally it may be used to turn the cursor on or off.

ALPHABETICAL REFERENCE GUIDE

LOCATE Statement

Rules:

1. Any values entered outside of these ranges will result in an `Illegal Function Call` error. Previous values are retained.

2. Any parameter may be omitted. Omitted parameters assume the old value.

Example:

`10 LOCATE 1,1`	Moves to the home position in the upper left hand corner.
`20 LOCATE ,,1`	Make the blinking cursor visible, position remains unchanged.
`30 LOCATE`	Position and cursor visibility remain unchanged.
`40 LOCATE 5,1,1`	Move to line five, column one, turn cursor on.

LOF Function

BRIEF

Format: `LOF(<file number>)`

Purpose: Returns the length of the file in bytes.

Details

The LOF function returns the length of the file in bytes. This command is also used in random files to determine the last record number of the file. LOF divided by the length of a record is equal to the number of records in the file.

Example:

```
10 OPEN "R",1,"PARTS",128
20 FIELD #1,128 AS DESC$
30 INPUT "ENTER PART# TO EXAMINE"; PN
40 IF PN <=0 THEN END
50 IF PN > LOF(1)/128 THEN PRINT "BAD REQUEST": GOTO 30
60 GET #1, PN
70 PRINT "DESCRIPTION:"; DESC$
80 GOTO 30
```

LOG Function

BRIEF

Format: `LOG(X)`

Action: Returns the natural logarithm of X. X must be greater than zero.

Details

Example:

```
PRINT LOG(45/7)
 1.860752
Ok
```

LPOS Function

BRIEF

Format: `LPOS(X)`

Action: Returns the current position of the line printer print head within the line printer buffer.

Details

The LPOS function does not necessarily give the physical position of the print head. X is a dummy argument.

Example:

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

LPRINT and LPRINT USING Statements

BRIEF

Format: `LPRINT [<list of expressions>]`
`LPRINT USING <string exp>;<list of expressions>`

Purpose: To print data at the line printer.

Details

The LPRINT statement is the same as PRINT and PRINT USING, except output goes to the line printer. See Pages 10.130 — 10.135.

LPRINT assumes a 255-character-wide printer.

LSET and RSET Statements

BRIEF

Format: `LSET <string variable> = <string expression>`
`RSET <string variable> = <string expression>`

Purpose: To move data from memory to a random file buffer.

Details

If <string expression> requires fewer bytes than were fielded to <string variable>, LSET left-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. RSET right-justifies the string. If the characters are too long for the field, RSET drops characters from the left. Numeric values must be converted to the strings before they are LSET or RSET. See the MKI$, MKS$, and MKD$ functions, Page 10.107.

Examples:

```
150 LSET A$=MKS$(AMT)
160 LSET D$=DESC$
```

See also Chapter 6, "File Handling," Pages 6.21, 6.22.

LSET or RSET may also be used with a non-fielded string variable to left-justify or right-justify a string in a given field. For example, the program lines:

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right-justify the string N$ in a 20-character field. This can be very useful for formatting printed output.

MERGE Command

BRIEF

Format: `MERGE <filename>`

Purpose: To merge a specified disk file into the program currently in memory.

Details

<filename> is the name used when the file was saved. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to Chapter 2, Page 2.12 for information about possible filename extensions under the Z-DOS Operating System.) The file must have been saved in ASCII format. (If not, a `Bad file mode` error occurs.)

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (Merging may be thought of as "inserting" the program lines on disk into the program in memory.)

BASIC always returns to command level after executing a MERGE command.

Example:

```
MERGE "NUMBERS"
```

MID$ Function

BRIEF

Format: `MID$(X$,I[,J])`

Action: Returns a string of length J from X$ beginning with the Ith character.

Details

I must be in the range one to 255. The range of J is from zero to 255. If J is omitted, or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I>LEN(X$), or J = 0. MID$ returns a null string.

Example:

```
10 A$="GOOD"
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,8,8)
RUN
GOOD EVENING
Ok
```

Also see LEFT$, Page 10.85 and RIGHT$, Page 10.150.

If I=0 is specified, the error message `Illegal Function Call in <linenumber>` will be returned.

MID$ Statement

BRIEF

Format: `MID$(<stringexpl>,n[,m])=<string exp2>`

where n and m are integer expressions and <string exp1> and <string exp2> are string expressions.

Purpose: To replace a portion of one string with another string.

Details

The characters in <string exp1>, beginning at position n, are replaced by the characters in <string exp2>. The optional m refers to the number of characters from <string exp2> that will be used in the replacement. If m is omitted, all of <string exp2> is used. However, regardless of whether m is omitted or included, the replacement of characters never goes beyond the original length of <string exp1>.

Example:

```
10 A$="KANSAS CITY, MO"
20 MID$(A$,14)="KS"
30 PRINT A$
RUN
KANSAS CITY, KS
Ok
```

MID$ is also a function that returns a substring of a given string.

MKI$, MKS$, MKD$ Functions

BRIEF

Format: `MKI$(<integer expression>)`
`MKS$(<single precision expression>)`
`MKD$(<double precision expression>)`

Action: Convert numeric values to string values.

Details

Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a two-byte string. MKS$ converts a single-precision number to a four-byte string. MKD$ converts a double-precision number to an eight-byte string.

Example:

```
90 AMT=K+T
100 FIELD #1,8 AS D$, 20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ =A$
130 PUT #1
      .
      .
```

See also CVI, CVS, CVD, Page 10.30 and Chapter 6, "File Handling."

NAME Command

BRIEF

Format: `NAME <old filename> AS <new filename>`

Purpose: To change the name of a disk file.

Details

The <old filename> must exist and <new filename> must not exist; otherwise an error will result. After a NAME command, the file exists on the same disk, in the same area of disk space, with the new name.

Example:

```
Ok
NAME "ACCTS" as "LEDGER"
Ok
```

NOTE: NAME does not assume .BAS extension.

NEW Command

BRIEF

Format: `NEW`

Purpose: To delete the program currently in memory and clear all variables.

Details

NEW is entered at command level to clear memory, closes all files and turns trace off before entering a new program. BASIC always returns to command level after a NEW command is executed.

NULL Statement

BRIEF

Format: `NULL <integer expression>`

Purpose: To set the number of nulls to be printed at the end of each line.

Details

For 10-character-per-second tape punches, <integer expression> should be >=three. When tapes are not being punched, <integer expression> should be zero or one for Teletypes and Teletype-compatible terminal screens. <integer expression> should be two or three for 30 cps hard copy printers. The default value is zero. The range is between zero and 255.

Example:

```
Ok
NULL 2
Ok
100 INPUT X
200 IF X<50 GOTO 800
      .
      .
      .
```

Two null characters will be printed after each line.

OCT$ Function

BRIEF

Format: `OCT$(X)`

Action: Returns a string which represents the octal value of the decimal argument.

Details

X is rounded to an integer before OCT$(X) is evaluated.

Example:

```
PRINT OCT$(24)
 30
Ok
```

See the HEX$ function for hexadecimal conversion, Page 10.64.

ON ERROR GOTO Statement

BRIEF

Format: `ON ERROR GOTO <line number>`

Purpose: To enable error trapping and specify the first line of the error handling subroutine.

Details

Once error trapping has been enabled all errors detected, including direct mode errors (e.g., syntax errors), will cause a jump to the specified error handling subroutine. If <line number> does not exist, an `Undefined line number` error results. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution.

An `ON ERROR GOTO 0` statement that appears in an error trapping subroutine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an `ON ERROR GOTO 0` if an error is encountered for which there is no recovery action.

If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

Example:

```
10 ON ERROR GOTO 80
20 INPUT "Enter number 1";N1
30 INPUT "Enter number 2";N2
40 A=N1/N2
50 B=N1*N2
60 PRINT A,B
70 GOTO 20
80 IF ERR=11 THEN PRINT"Do not enter zero for number 2!":RESUME 30
90 IF ERR=6 THEN PRINT"Do not enter such large numbers!":RESUME 20
100 PRINT"Error has occured.  It is error number:";ERR
```

Line 10 is the statement that tells BASIC where to go in the event of an error. In lines 20 and 30 the input statements ask for two numbers to be entered. In line 40 the first number (N1) is divided by the second number (N2) and the result is assigned to variable A. In line 50, the numbers are multiplied together and the result is assigned to variable B. Both A and B are then printed on the screen (line 60).

If you input a zero for the second number you will cause an error condition, and the program goes to line 80. Line 80 says if error number 11 occurs, which is BASIC's division by zero error (see Appendix A), then print "Do not enter zero for the number 2!" Line 90 says if error 6 occurs, which is the overflow error, then print "Do not enter such large numbers".

ON...GOSUB and ON...GOTO Statements

BRIEF

Format: `ON <expression> GOTO <list of line numbers>`
`ON <expression> GOSUB <list of line numbers>`

Purpose: To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Details

The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

If the value of <expression> is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of <expression> is negative or greater than 255, an `Illegal Function Call` error occurs.

Example:

```
100 ON L-1 GOTO 150,300,320,390
```

OPEN Statement

BRIEF

Format: `OPEN<"mode">,<#><file number>,<filename>,[<reclen>]`

Purpose: To allow I/O to a disk file.

Details

A disk file must be opened before any disk I/O operation can be performed on that file. The OPEN statement allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

<mode> is a string expression whose only character is one of the following:

O	specifies sequential output mode
I	specifies sequential input mode
R	specifies random input/output mode

<file number> is an integer expression whose value is between one and 255. The number is then associated with the file for as long as it is open and is used to refer other disk I/O statements to the file.

<filename> is a string expression containing a name that conforms to your operating system's rules for disk filenames. You may also need to specify a device name if the file you are opening is not on the default drive.

<reclen> is an integer expression which, if included, sets the record length for random files. The default record length is 128 bytes.

A file can be opened for sequential input or random access on more than one file number at a time. A file may be opened for sequential output, however, on only one file number at a time.

Example:

```
10 OPEN "I",2,"INVEN"
```

This program opens a sequential file called "INVEN" on unit two.

Also see "File Handling" (Page 6.1).

ALPHABETICAL REFERENCE GUIDE

OPEN Statement

BRIEF

Format: `OPEN [<dev>] <filename>[FOR <mode>] AS <#>`
`<file number> [LEN=<lrecl>]`

Purpose: To establish communication between a physical device and an I/O buffer in the data pool.

Details

<dev> is optionally part of the filename string and may be one of the following:

`A:-D:`	for Disk
`KYBD:`	Keyboard — Input Only
`LPT1:`	Printer — Output Only
`SCRN:`	Screen — Output Only
`COM1:`	RS-232 Communications 1

<filename> Is a valid string literal or variable optionally containing a <dev>. If <dev> is omitted, the default disk is assumed. Refer to "DISK FILES" for naming conventions.

<mode> Determines the initial positioning within the file and the action to be taken if the file does not exist. The valid modes and actions taken are:

INPUT — Position to the beginning of an existing file. A `File not found` error is given if the file does not exist.

OUTPUT — Position to the beginning of the file. If the file does not exist, one is created.

APPEND — Position to the end of the file. If the file does not exist, one is created.

If the FOR <mode> clause is omitted, the initial position is at the beginning of the file. If the file is not found, one is created. This is the random I/O mode. That is, records may be read or written at will at any position within the file.

<file number> Is an integer expression returning a number in the range one thru 255. The number is used to associate an I/O buffer with a disk file or device. This association exists until a CLOSE or CLOSE <file number> statement is executed.

lrecl Is an integer expression in the range one to 65535. This value sets the record length to be used for random files (see the FIELD statement). If omitted, the record length defaults to 128 byte records.

Action:

For each device, the following OPEN modes are allowed:

KYBD:	INPUT only.
SCRN:	OUTPUT only.
COM1:	INPUT, OUTPUT or random only.
LPT1:	OUTPUT only.

Disk files allow all modes.

When a disk file is opened FOR APPEND, the position is initially at the end of the file and the record number is set to the last record of the file (LOF(x)/128). PRINT, WRITE or PUT will then expand the file. The Program may position elsewhere in the file with a GET statement. If this is done, the mode is changed to random and the position moves to the record indicated.

OPEN Statement

Once the position is moved from the end of the file, additional records may be appended to the file by executing a GET #x,LOF(x)/<lrecl> statement. This positions the file pointer at the end of the file in preparation for appending.

Rules:

1. Any values entered outside of the ranges given will result in an `Illegal Function Call` error. The file is not opened.

2. If the file is opened as INPUT, attempts to write to the file will result in a `Bad File Mode` error.

3. If the file is opened as OUTPUT, attempts to read the file will result in a `Bad File Mode` error.

4. At any one time, it is possible to have a particular disk filename open under more than one file number. This allows different modes to be used for different purposes. Or, for program clarity, to use different file numbers for different modes of access. Each file number has a different buffer, so several records from the same file may be kept in memory for quick access.

   A file may **not** be opened FOR OUTPUT, on more than one file number at a time.

Example:

```
10 OPEN "PARTS.DAT" AS #1 'for random I/O on Disk A:

10 OPEN "KYBD:" FOR INPUT AS #2

10 OPEN "B:INVENT.DAT" FOR APPEND AS #1
```

OPEN COM Statement

BRIEF

Format: `OPEN "DEV:<speed>,<parity>,<data>,<stop>" AS [#]<file number>`

Function: OPEN "COM..." allocates a buffer for I/O in the same fashion as OPEN for disk files.

Details

OPENING A COM FILE

This section describes the BASIC statements required to support RS-232 asynchronous communication with other computer and peripherals.

DEV: Is a valid communications device. The valid device is COM1:

<speed> Is a literal integer specifying the transmit/receive baud rate. Valid speeds are: 75, 110, 150, 300, 600, 1200, 1800, 2400, 4800, 9600.

<parity> Is a one character literal specifying the parity for transmit and receive as follows:

S	SPACE,	Parity bit always transmitted and received as space (0 bit).
O	ODD,	Odd transmit/receiver parity checking.
M	MARK,	Parity bit always transmitted and received as mark (1 bit).
E	EVEN,	Even transmit/receive parity checking.
N	NONE,	No transmit parity, no receive parity checking.

<data> Is a literal integer indicating the number of transmit/receive data bits. Valid values are: 4,5,6,7, or 8.

Parity is a method by which data is checked to make sure it hasn't changed during transmission.

When odd parity is used, a parity bit is sent along with each character that is sent to the I/O device. Before transmission, this bit is set to either one or zero to ensure that the sum of all of the transmitted bits is an odd number. If the I/O device receives a byte of data bits and a parity bit that do not all add up to an odd number, then an error must have occurred during transmission.

NOTE: Four data bits with no parity is illegal. Also, eight data bits with any parity is illegal.

<stop> Is a literal integer indicating the number of stop bits. Valid values are: 1 or 2. If omitted then 75 and 110 bps transmit two stop bits, all other transmit one stop bit.

<file number> Is an integer expression returning a valid file number. The number is then associated with the file for as long as it is open and is used to refer other COM I/O statements to the file.

Missing parameters invoke the following defaults:

Speed — 300 bps
Parity — Even
Bits — 7

NOTE: A COM device may be opened to only one file number at a time.

Possible Errors:

Any coding errors within the filename string will result in a `Illegal Filename` error. An indication as to which parameter is in error will not be given.

A `Device Timeout` error will occur if Data Set Ready (DSR) is not detected. Refer to hardware documentation for proper cabling instructions.

Example:

```
10 OPEN "COM1: " AS #1
```

File one is opened for communication with all defaults. Speed at 300 bps, even parity, and seven data bits, one stop bit.

```
20 OPEN "COM1:2400 " AS #2
```

File two is opened for communication at 2400 bps. Parity and number of data bits are defaulted.

```
10 OPEN "COM1:1200,N,8" AS #1
```

File number one is opened for asynchronous I/O at 1200 bps, no parity is to be produced or checked, and eight bit bytes will be sent and received.

For more information concerning communication I/O, see Appendix F.

OPTION BASE Statement

BRIEF

Format: `OPTION BASE n`
where n is 1 or 0

PURPOSE: To declare the minimum value for array subscripts.

Details

The OPTION BASE statement is used to declare the minimum value for array subscripts. The default base is 0. This may be changed to 1. The OPTION BASE statement must be executed before any DIM statement is executed. If an OPTION BASE statement appears after an array has been dimensioned, a `Duplicate Definition` error will result. If the statement:

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is one.

OUT Statement

BRIEF

Format: `OUT I,J`
where I is an integer expression in the range −32768 — 65535.
J is an integer expression in the range zero to 255.

Purpose: To send a byte to a machine output port.

Details

The OUT statement is used to send a byte to a machine output port. The integer expression I is the port number, and the integer expression J is the data to be transmitted.

Example:

```
100 OUT 32,100
```

In this example, the value 100 is sent to port 32.

PAINT Statement

BRIEF

Format: `PAINT (Xstart,Ystart)[,paint attribute [,border attribute]]`

Purpose: To fill a graphics figure of the specified border at the specified border attribute with the fill attribute.

Details

The PAINT statement will fill in an arbitrary graphics figure of the specified border attribute with the specified fill attribute. The paint attribute will default to the foreground attribute if not given, and the border attribute defaults to the paint attribute.

For example, you might want to fill in a circle of attribute one with attribute two. Visually, this could mean a blue ball with a green border.

PAINT must start on a non-border point, otherwise PAINT will have no effect.

PAINT can fill any figure, but painting "jagged" edges or very complex figures may result in an `Out of Memory` error. If this happens, you must use the CLEAR statement to increase the amount of stack space available.

PEEK Function

BRIEF

Format: `PEEK(I)`

Action: Returns the byte (decimal integer in the range zero to 255) read from memory location I.

Details

I must be in the range −32768 to 65536. PEEK is the complementary command to the POKE function on Page 10.127.

Example:

```
A=PEEK(&H5A00)
```

POINT Function

BRIEF

Format: `POINT (X,Y)`

Function: Allows the user to read the attribute value of a pixel from the screen.

Details

The POINT function allows you to read the color value of a pixel from the screen. If the point given is out of range, the value negative one is returned. Valid returns are any integer between zero and seven.

Example:

```
10 FOR C=0 TO 7
20 PSET (10,10) ,C
30 IF POINT(10,10)<>C THEN PRINT
   "Black and white computer!"
50 NEXT C


10 IF POINT (i,i)<>0 THEN PRESET (i,i) ELSE PSET (i,i)
'invert current state of a point
```

For further information on the POINT function, see Chapter 7.

Poke Function

BRIEF

Format: `POKE I,J`
where I and J are integer expressions

Action: Writes a byte into a memory location.

Details

The POKE function will change the contents of a memory location. The integer expression I is the address of the memory location to be changed. The integer expression J is the value to be placed into memory location I. J must be in the range 0 to 255. I must be in the range −32768 to 65535.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. See Page 10.124.

POKE and PEEK are useful for efficient data storage, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

Example:

```
10 POKE 34000,1
```

This example places the value one in memory location 34000.

WARNING: The POKE function should only be used by experienced users who know exactly what they are doing. It is possible to damage or destroy important data located in memory by using this function in the wrong way.

POS Function

BRIEF

Format: `POS(I)`

Action: Returns the current cursor position.

Details

The POS function will return the current cursor position. The leftmost position is 1. I is a dummy argument.

Example:

```
IF POS(I) >60 THEN PRINT CHR$(13)
```

PRESET Statement

BRIEF

Format 1: `PRESET (Xcoordinate , Y coordinate) [,attribute ]`

Format 2: `PRESET STEP (X offset, Y offset)[,attribute]`

Purpose: To turn off a point on the screen at a specified location.

Details

PRESET has an identical syntax to PSET. The only difference is that if no third parameter is given, the background color — zero is selected. When a third argument is given, PRESET is identical to PSET.

Example:

```
10 FOR I=0 to 100
20 PSET (I,I)
30 NEXT
   (draw a diagonal line to (100,100))
40 FOR I=100 TO 0 STEP -1
50 PRESET (I,I)
60 NEXT
```

Notice that in the preceding example is the same example given for PSET on Page 10.139. The only difference is in line 50;

```
50 PRESET (I,I)
```

Notice there is no third parameter given. The PRESET statement causes all of the specified points to be turned on to the background color. If a color argument was added to this line, the effect would be the same as using PSET.

If an out of range coordinate is given to PSET or PRESET, no action is taken nor is an error given. If an attribute greater than seven is given, this will result in an illegal function call.

For further information on PRESET, see Chapter 7.

PRINT Statement

BRIEF

Format: `PRINT (<List of Expressions>)`

Purpose: To output data at the terminal.

Details

If <list of expressions> is omitted from a PRINT statement, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

PRINT POSITIONS

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. A question mark may be used in place of the word PRINT in a PRINT statement.

Example 1:

```
10 X=5
20 PRINT X+5, X-5, X*(-5), X^5
30 END
RUN
 10             0            -25            3125
Ok
```

In Example 1, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2:

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
RUN
?9
 9 SQUARED IS 81 AND 9 CUBED IS 729

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

In Example 2, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

Example 3:

```
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
RUN
 5   10   10   20   15   30   20   40   25   50
Ok
```

In Example 3, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

Additional Considerations

Single-precision numbers that can be represented with seven or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format are output using the unscaled format. For example, 1E-7 is output as .0000001, and 1E-8 is output as 1E-08. Double-precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format are output using the unscaled format. For example, 1D-16 is output as .0000000000000001, and 1D-17 is output as 1D-17.

PRINT USING Statement

BRIEF

Format: `PRINT USING, <string exp>; <list of expressions>`

Purpose: To print strings or numbers using a specified format.

Details

<list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons or commas. <string exp> is a string literal (or variable) comprised of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

STRING FIELDS

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!"	Specifies that only the first character in the given string is to be printed.
"\n spaces\"	Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.

Example:

```
10 A$="Hello":B$="you"
20 PRINT USING "\\!";A$,B$
30 PRINT USING "\   \ \ \";A$,B$
RUN
Hey
Hello you
```

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

Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
Ok
```

NUMERIC FIELDS

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

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

. A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

Example:

```
PRINT USING "##.##";.78
 0.78
Ok

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

PRINT USING "##.##    ";10.2,5.3,66.789,.234
10.20    5.30    66.79    0.23
Ok
```

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

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

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

PRINT USING Statement

```
PRINT USING "+##.##    ";-68.95,2.4,55.6,-.9
-68.95     +2.40    +55.60     -0.90
Ok

PRINT USING "##.##-    ";-68.95,22.449,-7.01
68.95-    22.45     7.01-
Ok
```

**

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

```
PRINT USING "**#.#    ";12.39,-0.9,765.1
*12.4    *-0.9    765.1
Ok
```

$$

A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format can be used with $$. Negative numbers can also be used.

```
PRINT USING "$$###.##";1456.78
$1456.78
Ok
```

**$

The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

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

,

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

```
PRINT USING "####,.##,";1234.5
1,234.50
Ok

PRINT USING "####.##,";1234.5
1234.50,
Ok
```

^^^^

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

```
PRINT USING "##.##^^^^";234.56
 2.35E+02
Ok

PRINT USING ".####^^^^-";888888
.8889E+06
Ok

PRINT USING "+.##^^^^";123
+.12E+03
Ok
```

_

An underscore in the format string causes the next character to be output as a literal character.

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

You may print the underscore as a literal character itself by placing "__" in the format string.

%

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

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

PRINT USING ".##";.999
%1.00
Ok
```

If the number of digits specified exceeds 24, an `Illegal Function Call` error will result.

PRINT# and PRINT# USING Statements

BRIEF

Format: `PRINT#<filenumber>,[USING<stringexp>;]<listofexps>`

Purpose: To write data to a sequential disk file.

Details

<file number> is the number used when the file was opened for output. <string exp> is comprised of formatting characters as described in PRINT USING. The expressions in <list of expressions> are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the terminal screen with a PRINT statement. For this reason, care should be taken to delimit the data on the disk so that it will be input correctly from the disk.

In a list of expressions, numeric expressions should be delimited by semicolons or commas.

Example:

```
PRINT#1,A,B,C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to disk.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

Example:

```
A$="CAMERA":B$="93604-1".
```

The statement:

```
PRINT#1,A$;B$
```

would write CAMERA93604-1 to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT#1,A$;",";B$
```

The image written to disk is:

CAMERA,93604-1

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to disk surrounded by explicit quotation marks, CHR$(34).

Example:

```
100 A$="FRANK, RICHARD"
110 PRINT #1, CHR$(34)+A$+CHR$(34)
```

Since the data written to the disk contains a comma, it has been explicitly surrounded by quotation marks (CHR$(34)). The statement, INPUT #1, N$ would read in the complete data item — FRANK, RICHARD.

The PRINT# statement may also be used with the USING option to control the format of the disk file.

Example:

```
PRINT#1,USING "$$###.##,";J;K;L
```

PSET Statement

BRIEF

Format 1: `PSET (X coordinate , Y coordinate) [,attribute ]`

Format 2: `PSET STEP (X offset, Y offset) [,attribute]`

Purpose: To turn on a point at a specified location on the screen.

Details

The first argument to PSET is the coordinate of the point that you wish to plot. Coordinates always can come in one of two forms:

```
STEP (X offset, Y offset) or
( absolute X, absolute Y)
```

The first form is a point relative to the most recent point referenced. The second form is more common and directly refers to a point without regard to the last point referenced.

```
(10,10) absolute form
STEP (10,0) offset 10 in X and 0 in Y
(0,0) origin
```

When BASIC scans coordinate values it will allow them to be beyond the edge of the screen, however values outside the integer range (−32768 to 32767) will cause an overflow error.

(0,0) is always the upper left hand corner. It may seem strange to start numbering Y at the top so that the bottom left corner is (0,224), but this is standard.

It is not necessary to specify the color argument to PSET. If attribute is omitted then the default value is one, since this is the foreground attribute.

Example:

```
 5 CLS
10 FOR I=0 to 100
20 PSET (I,I)
30 NEXT
  (draw a diagonal line to (100,100))
40 FOR I=100 TO 0 STEP -1
50 PSET (I,I),0
60 NEXT
   (clear out the line by setting each pixel to 0)
```

For more information concerning the PSET statement, see Chapter 7.

PUT Statement

BRIEF

Format: `PUT <#><file number>[,<record number>]`

Purpose: To write a record from a random buffer to a random disk file.

Details

<file number> is the number under which the file was OPENed. If <record number> is omitted, the record will have the next available record number (after the last PUT). The largest possible record number is 32767. The smallest record number is 1.

See Pages 6.22 – 6.23.

PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before a PUT statement.

In the case of WRITE#, BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a `Field overflow` error.

RANDOMIZE Statement

BRIEF

Format: `RANDOMIZE [<expression>]`

Purpose: To reseed the random number generator.

Details

The RANDOMIZE statement is used to reseed the random number generator. <expression> is used as the random number seed value. If <expression> is omitted, BASIC suspends program execution and asks for a value by printing:

```
Random Number Seed (-32768 to 32767)?
```

The value input is used as the random number seed.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers every time the program is run, place a RANDOMIZE statement at the beginning of the program and change the argument with each run.

Example:

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)? 3   (user types 3)

 .88598   .484668   .586328   .119426   .709225
Ok

RUN
Random Number Seed (-32768 to 32767)? 4   (user types 4 for new sequence)
 .803506   .162462   .929364   .292443   .322921
Ok

RUN
Random Number Seed (-32768 to 32767)? 3   (same sequence as first run)
 .88598   .484668   .586328   .119426   .709225
Ok
```

Note: These numbers may vary.

READ Statement

BRIEF

Format: `READ <list of variables>`

Purpose: To read values from DATA statements and assign them to variables. (See DATA, Page 10.31.)

Details

A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a `Syntax error` will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an `Out of DATA` message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (see RESTORE, Page 10.147).

Example 1:

```
        .
        .
        .
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
        .
        .
        .
```

This program segment reads the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2:

```
10 PRINT "CITY", "STATE", "ZIP"
20 READ C$,S$,Z
30 DATA "DENVER", "COLORADO", 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY          STATE         ZIP
DENVER       COLORADO      80211
Ok
```

This program reads string and numeric data from the DATA statement in line 30.

REM Statement

BRIEF

Format: `REM [<remark>]`

Purpose: To allow explanatory remarks to be inserted in a program.

Details

REM statements are not executed, but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

You may add remarks to the end of a line by preceding the remark with a single quotation mark instead of REM.

WARNING: Do not use this in a data statement, as it would be considered legal data.

Example:

```
        .
        .
        .
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
        .
        .
        .
```

or:

```
        .
        .
        .
120 FOR I=1 TO 20     'CALCULATE AVERAGE VELOCITY
130 SUM=SUM+V(I)
140 NEXT I
        .
        .
        .
```

RENUM Command

BRIEF

Format: `RENUM [<new number>][,<old number>][,<increment>]`

Purpose: To renumber program lines.

Details

The RENUM command is used to automatically renumber program lines. <new number> is the first line number to be used in the new sequence. The default is 10. <old number> is the line in the current program where renumbering is to begin. The default is the first line of the program. <increment> is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON. . .GOTO,ON. . .GOSUB and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message `Undefined line xxxxx in yyyyy` is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy may be changed.

RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An `Illegal Function Call` error will result.

Examples:

`RENUM`	Renumbers the entire program. The first new line number will be 10. Lines will increment by 10.
`RENUM 300,,50`	Renumbers the entire program. The first new line number will be 300. Lines will increment by 50.
`RENUM 1000,900,20`	Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

RESET Command

BRIEF

Format: `RESET`

Purpose: To close all disk files and write the directory information to a disk before it is removed from a disk drive.

Details

Always execute a RESET command before removing a disk from a disk drive. Otherwise, when the diskette is used again, it will not have the current directory information written on the directory track.

RESET closes all open files on all drives and writes the directory track to every disk with open files.

RESTORE Statement

BRIEF

Format: `RESTORE [<line number>]`

Purpose: To allow DATA statements to be reread from a specified line.

Details

After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the first DATA statement at or following <line number>.

Example:

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
      .
      .
      .
```

RESUME Statement

BRIEF

Formats: `RESUME`

`RESUME 0`

`RESUME NEXT`

`RESUME <line number>`

Purpose: To continue program execution after an error recovery procedure has been performed.

Details

Any one of the four formats shown above may be used, depending upon where execution is to resume:

`RESUME` or `RESUME 0`	Execution resumes at the statement which caused the error.
`RESUME NEXT`	Execution resumes at the statement immediately following the one which caused the error.
`RESUME <line number>`	Execution resumes at <line number>.

A RESUME statement that is not in an error trap routine causes a `RESUME without error` message to be printed.

Example:

```
10 ON ERROR GOTO 900
      .
      .
      .
900  IF  (ERR=230) AND (ERL=90)  THEN  PRINT  "TRY
AGAIN":RESUME 80
      .
      .
      .
```

RETURN Statement

BRIEF

Format: `RETURN <line number>`

Purpose: To allow the use of a non-local return for event trapping.

Details

This optional form of RETURN is primarily intended for use with event trapping. The event trap routine may want to go back into the BASIC program at a fixed line number while still eliminating the GOSUB entry that the trap created.

Use of the non-local RETURN must be done with care! Any other GOSUB, WHILE or FOR that was active at the time of the trap will remain active. If the trap comes out of a subroutine, any attempt to continue loops outside the subroutine will result in a `NEXT without FOR` error.

See the GOSUB...RETURN statement on Page 10.62 for a discussion of normal use of RETURN.

RIGHT$ Function

BRIEF

Format: `RIGHT$(X$,I)`

Action: Returns the right-most I characters of string X$.

Details

The RIGHT$ function will return the right-most I characters of string X$. If I is greater than or equal to the length of the string X$, the function will return the entire string. If I=0, the null string (length zero) is returned. I must be in the range of zero to 255.

Example:

```
10 A$="DISK BASIC"
20 PRINT RIGHT$(A$,5)
RUN
BASIC
Ok
```

Also see the MID$ and LEFT$ functions.

RND Function

BRIEF

Format: `RND(X)`

Action: Returns a random number between 0 and 1.

Details

The RND function returns a random number between 0 and 1. The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded (see RANDOMIZE). However, X<0 always restarts the same sequence for any given X.

X>0 or X omitted generates the next random number in the sequence. X=0 repeats the last number generated.

Example:

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT I
RUN

24   30   31   51   5
Ok
```

NOTE: The RND function with no argument specified is the same as RND with a positive argument.

RUN Command

BRIEF

Format 1: `RUN [<line number>]`

Format 2: `RUN <filename>[,R]`

Purpose: To execute the program currently in memory, or (format 2) to load a file from disk into memory, and run it.

Details

The RUN command is used to execute the program currently in memory. If <line number>is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. BASIC always returns to command level after a RUN is executed.

In format 2, <filename> is the name used when the file was saved. (Your operating system may append a default filename extension if one was not supplied in the SAVE command.)

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain open.

Example:

```
RUN "NEWFIL",R
```

The BASIC Compiler supports both the RUN and RUN <line number > forms of the RUN command. The BASIC Compiler does not support the "R" option with RUN. If you want this feature, use the CHAIN statement.

SAVE Command

BRIEF

Format: `SAVE <filename>[,A|,P]`

Purpose: To save a program file on disk.

Details

The SAVE command is used to save a program file on a disk. <filename> is a quoted string that conforms to your operating system's requirements for filenames. Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to your Z–DOS Manual for information about possible filename extensions under the Z-DOS operating system. If <filename> already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access operations or procedures requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some operating system commands such as LIST may require an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later run (or loaded), any attempt to LIST or EDIT it will fail.

Examples:

```
SAVE"COM1",A
SAVE"PROG",P
```

SCREEN Function

BRIEF

Format: `X = SCREEN(row,col[,z])`

Function: The SCREEN Function returns the ordinal of the character from the screen at the specified row (line) and column.

Details

x Is a numeric variable receiving the ordinal returned.

row Is a valid numeric expression returning an unsigned integer in the range one to 25.

col Is a valid numeric expression returning an unsigned integer in the range one to 80.

z Is a valid numeric expression returning a Boolean result.

Action:

The ordinal of the character at the specified coordinates is stored in the numeric variable. If the optional parameter <z> is given and non-zero, the color attribute for the character is returned instead.

NOTE: Any values entered outside of these ranges will result in an `Illegal Function Call` error.

Example:

```
100 X = SCREEN (10,10) 'If the character at 10,10 is
                       'A then return 65.

110 X = SCREEN (1,1,1) 'Return the color attribute of
                       'the character in the upper left
                       'hand corner of the screen.
```

SCREEN Statement

BRIEF

Format: `Screen [graphics,] [reverse video]`

Purpose: The SCREEN statement sets the screen attributes.

Details

The SCREEN statement allows you to put H-19 graphic characters on the video display and also permits the use of reverse video.

Graphics is a numeric expression with the value of zero or one.

Reverse video is a numeric expression with the value of zero or one.

Graphics	0 — Clears H-19 Graphics mode 1 — Sets H-19 Graphics mode
Reverse Video	0 — Clears H-19 reverse video 1 — Sets H-19 reverse video

Action:

If all parameters are legal, the new screen mode is stored. If the new screen mode is the same as the previous mode, nothing is changed.

Rules:

1. Any values entered outside of these ranges will result in an `Illegal Function Call` error. Previous values are retained.

2. Any parameter may omitted. Omitted parameters assume the old value.

For further information concerning the SCREEN statement, see Chapter 7.

Example:

```
10 SCREEN 0,1       'No graphics, reverse video on
20 SCREEN 1         'Switch to H-19 graphics mode.
40 SCREEN 1,1       'Switch to H-19 graphics
                    with reverse video on.
50 SCREEN ,0        'graphics off and reverse video off.
```

SGN Function

BRIEF

Format: `SGN(X)`

Action: Returns the mathematical sign value.

Details

If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns −1.

Example: The statement

```
ON SGN(X)+2 GOTO 100,200,300
```

branches to 100 if X is negative, to 200 if X is 0, and to 300 if X is positive.

SIN Function

BRIEF

Format: `SIN(X)`

Action: Returns the sine of X in radians.

Details

SIN(X) is caluclated in single precision. COS(X)=SIN (X+3.145159/2).

Example:

```
PRINT SIN(1.5)
 .9974951
Ok
```

SPACE$ Function

BRIEF

Format: SPACE$(X)

Action: Returns a string of spaces of length X.

Details

The expression X is rounded to an integer and must be in the range 0 to 255.

Example:

```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
 1
  2
   3
    4
     5
Ok
```

Also see the SPC function on Page 10.159.

SPC Function

BRIEF

Format: `SPC(I)`

Action: Prints I blanks on the terminal or printer.

Details

The SPC function may only be used with PRINT and LPRINT statements. I must be in the range −32768 to 65535. A';' is assumed to follow the SPC(I) function.

Example:

```
PRINT "OVER" SPC(15) "THERE"
OVER              THERE
Ok
```

Note: When this command is used on the screen, values greater than 80 wrap around to the beginning of the same line rather than going down to the next line. Thus, SPC(85) is the same as SPC(5).

Also see the SPACE$ function Page 10.158.

NOTE: Negative numbers are treated as zero.

SQR Function

BRIEF

Format: `SQR(X)`

Action: Returns the square root of X. X must be >=0.

Details

The SQR function returns the square root of X. X must be greater than or equal to 0.

Example:

```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10           3.162278
 15           3.872984
 20           4.472146
 25           5
Ok
```

Also see "Numeric Functional Operators", Page 5.46.

STOP Statement

BRIEF

Format: STOP

Purpose: To terminate program execution and return to command level.

Details

STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

```
Break in nnnnn
```

Unlike the END statement, the STOP statement does not close files.

BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command (see Page 10.25).

Example:

```
10 INPUT A,B,C
20 K=A^2*5.3:L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
Break in 30
Ok
PRINT L
 30.76923
Ok
CONT
 115.9
Ok
```

STR$ Function

BRIEF

Format: `STR$(X)`

Action: Returns a string representation of the value of X.

Details

The STR function is used to convert numbers to a string representation.

Example:

```
10 INPUT "TYPE A NUMBER";N
20 B$="Number entered was" + STR$(N)
30 PRINT B$
```

This example converts the number that is input to a string so that it can be attached to the sentence and placed in B$.

The VAL function is the inverse function of STR$.

STRING$ Function

BRIEF

Formats: `STRING$(I,J)`
`STRING$(I,X$)`

Action: Returns a string of length I whose characters all have ASCII code J or the first character of X$.

Details

The STRING$ function returns a string of length I whose characters all have ASCII code J or the first character of X$. See Appendix C for ASCII values.

Example:

```
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
Ok
```

SWAP Statement

BRIEF

Format: `SWAP <variable>,<variable>`

Purpose: To exchange the values of two variables.

Details

Any type variable may be swapped (integer, single-precision, double-precision, string), but the two variables must be of the same type or a `Type mismatch` error results.

Example:

```
10 A$="ONE" : B$="ALL" : C$=" FOR "
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
Ok
RUN
ONE FOR ALL
ALL FOR ONE
Ok
```

SYSTEM Command

BRIEF

Format: `SYSTEM`

Purpose: To exit BASIC and return to the operating system.

Details

The SYSTEM command closes all files, clears all variables, removes all programs from memory and returns to the operating system. The programs in memory should be saved prior to typing this command, or they will be lost if they are not already on the disk.

TAB Function

BRIEF

Format: `TAB(I)`

Action: Spaces to position I on the terminal.

Details

If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be in the range −32768 to 65535. TAB may only be used in PRINT and LPRINT statements.

Example:

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","$25.00"
RUN
NAME                      AMOUNT

G. T. JONES               $25.00
Ok
```

Note: When this command is used on the screen, values greater than 80 wrap around to the beginning of the same line rather than going to the next line. Thus, TAB(85) is the same as TAB(5).

TAN Function

BRIEF

Format: `TAN(X)`

Action: Returns the tangent of X in radians.

Details

TAN(X) is calculated in single-precision. If TAN overflows, the `Overflow` error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:

```
10 Y = Q*TAN(X)/2
```

TIME$ Statement

BRIEF

Format: TIME$ = <string expr> To set the current time.
<string var> = TIME$ To get the current time.

Purpose: The TIME$ statement may be used to set or retrieve the current time.

Details

<string expr> is a valid string literal or variable.

The current time is returned and assigned to the string variable if TIME$ is the expression in a LET or PRINT statement.

The current time is stored if TIME$ is the target of a string assignment.

Rules:

1. If <string expr> is not a valid string, a `Type mismatch` error will result.

2. For <string var>= TIME$, TIME$ returns an 8- character string in the form "hh:mm:ss", where hh is the hour (00 to 23), mm is the minutes (00 to 59), and ss is the seconds (00 to 59).

3. For TIME$ = <string expr>, <string expr> may be one of the following forms:

   A. "hh" Sets the hour. Minutes and seconds default to 00.

   B. "hh:mm:" Sets the hour and minutes. Seconds default to 00.

   C. "hh:mm:ss" Sets the hour, minutes, and seconds.

If any of the values are out of range, an `Illegal Function Call` error is issued. The previous time is retained.

Example:

```
TIME$ = "08:00"
Ok
PRINT TIME$
08:00:04
Ok
```

The following program displays the current date and time on the twenty-fifth line of the screen, and updates the displayed time every minute.

```
10 KEY OFF:CLS
20 LOCATE 25,5
30 PRINT DATE$, TIME$
40 T = TIME
50 IF TIME - T >59 THEN 20
60 GOTO 50
```

ALPHABETICAL REFERENCE GUIDE

TRON/TROFF Statements

BRIEF

Format: `TRON`
`TROFF`

Purpose: To trace the execution of program statements.

Details

As an aid in debugging, the TRON statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example:

```
10 K=10
20 FOR J=1 TO 2
30 L=K+10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
TRON
RUN
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
Ok
TROFF
Ok
```

USR Function

BRIEF

Format: `USR[<digit>](X)`

Action: Calls the user's assembly language subroutine with the argument X.

Details

<digit> is in the range zero to 9 and corresponds to the digit supplied with the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. See Appendix E, "BASIC Assembly Language Subroutines."

Example:

```
50 C = USR(B/2)
60 D = USR2(B/2)
     .
     .
     .
```

These two program lines call "user" programs that have been previously input to memory by the user.

See the DEF USR statement, Page 10.37

VAL Function

BRIEF

Format: `VAL(X$)`

Action: Returns the numerical value of string X$.

Details

The VAL function also strips leading blanks, tabs, and line feeds from the argument string. For example,

VAL(" −3")

returns −3.

Example:

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$)<60000 OR VAL(ZIP$)>60999 THEN
PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=60601 AND VAL(ZIP$)<=60699 THEN
PRINT NAME$ TAB(25) "IN TOWN"
      .
      .
      .
```

See the STR$ function for numeric to string conversion.

VARPTR Function

BRIEF

Format 1: `VARPTR(<variable name>)`

Format 2: `VARPTR(#<file number>)`

Action: Format 1: Returns the address of the first byte of data identified with <variable name>.

Format 2: For sequential files, returns the starting address of the disk I/O buffer assigned to <file number>.

Details

A value must be assigned to <variable name> prior to execution of VARPTR. Otherwise, an `Illegal Function Call` error results. Any type variable name may be used (numeric, string, array), and the address returned will be an integer in the range 32767 to −32768. If a negative address is returned, add it to 65536 to obtain the actual address.

The VARPTR function is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. Specify a function call of the form VARPTR(A(0)) when an array is passed, so that the lowest-addressed element of the array is returned.

Assign all simple variables before you call VARPTR for an array because the addresses of the arrays change whenever a new simple variable is assigned.

For random files, VARPTR returns the address of the FIELD buffer assigned to <file number>.

Example:

```
100 X=USR(VARPTR(Y))
```

ALPHABETICAL REFERENCE GUIDE

VARPTR Function

Format: `VARPTR(<file name>)`

Function: For files, the VARPTR function returns the address of the first byte of the File Control Block (FCB) for the opened file.

File number is tied to a currently open file. Offsets to information in the FCB from the address returned by VARPTR are:

OFF	SIZE	CONTENTS	
0	1	Mode	The mode in which the file was opened: 1 — Input Only 2 — Output Only 4 — Random I/O 16 — Append Only 32 — Internal use 64 — Future use 128 — Internal use
1	38	FCB	Disk File Control Block. Refer to Z–DOS User's Guide for Contents.
39	2	CURLOC	Number of sectors read or written for sequential access. For random access, it contains the last record number + 1 read or written.
41	1	ORNOFS	Number of bytes in sector when read or written.
42	1	NMLOFS	Number of bytes left in input buffer.
43	3	***	Reserved for future expansion.
46	1	DEVICE	Device Number: 0-9 - Disks A: thru J: 255 — KYBD: 254 — SCRN: 253 — LPT1: 251 — COM1:

47	1	WIDTH	Device width.
48	1	POS	Position in buffer for PRINT.
49	1	FLAGS	Internal use during LOAD/SAVE not used for data files.
50	1	OUTPOS	Output position used during tab expansions.
51	128	BUFFER	Physical data buffer. Used to transfer data between Z-DOS and BASIC. Use this offset to examine data in sequential I/O mode.
179	2	VRECL	Variable length record size. Default is 128. Set by length option in OPEN statement.
181	2	PHYREC	Current physical record number.
183	2	LOGREC	Current logical record number.
185	1	***	Future use.
186	2	OUTPOS	Disk files only. Output position for PRINT, INPUT and WRITE.
188	<n>	FIELD	Actual FIELD data buffer. Size is determined by length specified in OPEN statement. VRECL bytes are transferred between BUFFER and FIELD on I/O operations. Use this offset to examine file data in Random I/O mode.

Example:

```
10 OPEN "DATA.FIL" as #1
20 FCBADR = VARPTR(#1)    'FCBADR contains start of FCB.
30 DATADR = FCBADR+188  'DATADR contains address of data
                        buffer.
40 A$ = CHR$(PEEK DATADR)    'A$ contains lst byte in data
                             buffer.
```

WAIT Statement

BRIEF

Format: `WAIT <port number>, I[,J]`
where I and J are integer expressions

Purpose: To suspend program execution while monitoring the status of a machine input port.

Details

The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression J, and then AND'ed with I. If the result is zero, BASIC loops back and reads the data at the port again. If the result is non-zero, execution continues with the next statement. If J is omitted, it is assumed to be zero.

It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to manually restart the machine.

Example:

```
100 WAIT 32,2
```

WHILE...WEND Statement

BRIEF

Format:
```
WHILE <expression>
  .
  .
[<loop statements>]
  .
  .
WEND
```

Purpose: To execute a series of statements in a loop as long as a given condition is true.

Details

If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a `WHILE without WEND` error, and an unmatched WEND statement causes a `WEND without WHILE` error.

Example:

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115  FLIPS=0
120  FOR I=1 TO 10-1
130  IF A$(I)>A$(I+1) THEN
     SWAP A$(I),A$(I+1):FLIPS=1
140 NEXT I
150 WEND
```

WIDTH Statement

BRIEF

Format: `WIDTH <LPRINT><integer expression>`

Purpose: To set the printed line width in number of characters for the line printer.

Details

WIDTH LPRINT sets the line width at the line printer.

<integer expression> must have a value in the range one to 225. The only valid width for the terminal is 80 characters.

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

Example:

```
10 LPRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RUN
Ok
WIDTH LPRINT 18
Ok
RUN
Ok
```

This is what will appear on the printer.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQR
STUVWXYZ
```

WRITE Statement

BRIEF

Format: `WRITE[<list of expressions>]`

Purpose: To output data at the terminal.

Details

If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of the expressions are output at the terminal. The expressions in the list may be numeric and/or string expressions, and they must be separated by commas.

When the printed items are output, each item will be separated from the last by a comma. Printed strings will be delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement, Page 10.129.

Example:

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
80, 90, "THAT'S ALL"
Ok
```

WRITE # Statement

BRIEF

Format: WRITE#<file number>,<list of expressions>

Purpose: To write data to a sequential file.

Details

<file number> is the number under which the file was OPENed in "O" mode. The expressions in the list are string or numeric expressions, and they must be separated by commas or semicolons.

The difference between WRITE# and PRINT# is that WRITE # inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disk.

Example:

```
A$="CAMERA" and B$="93604-1".
```

The statement:

```
WRITE#1,A$,B$
```

writes the following image to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "93604 – 1" to B$.

APPENDIX A

Error Messages

SUMMARY OF ERROR CODES AND ERROR MESSAGES

Number	Message
1	**NEXT without FOR** A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
2	**Syntax error** A line is encountered that contains some incorrect sequence of characters (such as an unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.).
3	**RETURN without GOSUB** A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
4	**Out of DATA** A READ statement is executed when there are no DATA statements with unread data remaining in the program.
5	**Illegal function call** A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of: A. a negative or unreasonably large subscript B. a negative or zero argument with LOG C. a negative argument to SQR D. a negative mantissa with a non-integer exponent

APPENDIX A

Error Messages

Number	Message
	E. a call to a USR function for which the starting address has not yet been given
	F. an improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, ASC$ FN...( ) or ON...GOTO.
6	**Overflow** The result of a calculation is too large to be represented in BASIC's number format. If overflow occurs, the result is zero and execution continues without an error.
7	**Out of memory** A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.
8	**Undefined line number** A line reference in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE is to a nonexistent line.
9	**Subscript out of range** An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.
10	**Duplicate Definition** Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array.

APPENDIX A
Error Messages

Number	Message
11	**Division by zero** A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.
12	**Illegal direct** A statement that is illegal in direct mode is entered as a direct mode command.
13	**Type mismatch** A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.
14	**Out of string space** String variables have caused BASIC to exceed the amount of free memory remaining. BASIC will allocate string space dynamically, until it runs out of memory.
15	**String too long** An attempt is made to create a string more than 255 characters long.
16	**String formula too complex** A string expression is too long or too complex. The expression should be broken into smaller expressions.

Error Messages

Number	Message
17	**Can't continue** An attempt is made to continue a program that: A. has halted due to an error, B. has been modified during a break in execution, or C. does not exist.
18	**Undefined user function** A USR function is called before the function definition (DEF statement) is given.
19	**No RESUME** An error trapping routine is entered but contains no RESUME statement.
20	**RESUME without error** A RESUME statement is encountered before an error trapping routine is entered.
21	**Unprintable error** An error message is not available for the error condition which exists. This is usually caused by an error with an undefined error code.
22	**Missing operand** An expression contains an operator with no operand following it.
23	**Line buffer overflow** An attempt is made to input a line that has too many characters.

APPENDIX A

Error Messages

Number	Message
24	**Device Timeout** An attempt at I/O was made with a device that was not ready. After a given amount of time, this error message is produced. Check the device being called in the program line.
25	**Device Fault** An attempt at I/O was made with a device that has a problem. This error message may be caused by any number of conditions, from using the wrong diskette type to being out of paper. Check the device being called in the program line and correct the fault.
26	**FOR without NEXT** A FOR was encountered without a matching NEXT.
27	**Out of paper** If your printer can transmit error conditions via the parallel lines, this error condition can be detected. Check the printer and replace the paper.
29	**WHILE without WEND** A WHILE statement does not have a matching WEND.
30	**WEND without WHILE** A WEND was encountered without a matching WHILE.

Disk Errors

50	**FIELD overflow** A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.

APPENDIX A

Error Messages

Number	Message
51	**Internal error** An internal malfunction has occurred in BASIC. Report to Zenith the conditions under which the message appeared.
52	**Bad file number** A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.
53	**File not found** A LOAD, KILL or OPEN statement references a file that does not exist on the current disk.
54	**Bad file mode** An attempt is made to use PUT, or GET, with a sequential file, to LOAD a random file or to execute an OPEN with a file mode other than I, O, or R.
55	**File already open** A sequential output mode OPEN is issued for a file that is already open, or a KILL is given for a file that is open.
57	**Device I/O error** An I/O error has occurred on a device I/O operation. Check the device being called in the line where the error occurred.
58	**File already exists** The filename specified in a NAME statement is identical to a filename already in use on the disk.
61	**Disk full** All disk storage space is in use.

APPENDIX A
Error Messages

Number	Message
62	**Input past end** An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end of file.
63	**Bad record number** In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero.
64	**Bad file name** An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).
66	**Direct statement in file** A direct statement is encountered while LOADing an ASCII-format file. The LOAD is terminated.
67	**Too many files** An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.
68	**Device Unavailable** An attempt at I/O made with a device that is unavailable to the system.
69	**Communication buffer overflow** Your program has not properly maintained the communication buffer and has allowed it to fill up with data.
70	**Disk write protected** An attempt has been made to write to a disk that is write protected. Check the disk to ensure that it is the correct disk before you remove the write protect tab.

Error Messages

Number	Message
71	**Disk not Ready** This may be caused by the disk not being in the drive. Insert the disk and close the door.
72	**Disk Media Error** A fault has been discovered during a read/write operation, probably caused by a damaged disk.
73	**Advanced feature** An attempt was made to use a feature not available in this version of BASIC.
74	**Rename across disks** An attempt was made to rename a disk file specifying a device other than the one the file is on. Check the command for disk name continuity.

APPENDIX B

Converting Programs to Z-BASIC

BRIEF

If you have programs written in a BASIC other than Zenith BASIC, some minor adjustments may be necessary before running them with this version. Following are some specific things to look for when converting BASIC programs.

Details

String Dimension

Replace all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the Z-BASIC statement DIM A$(J).

Concatenation

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for Z-BASIC string concatenation.

Substring

Additionally, in this BASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings from strings. Forms such as A$(n) to access the nth character in A$, or A$(I,J) to take a substring of A$ from position I to J, must be changed as follows:

Other BASIC	Z-BASIC
`X$=A$(I)`	`X$=MID$(A$,I,1)`
`X$=A$(I,J)`	`X$=MID$(A$,I,J-I+1)`

If the string reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

Other BASIC	Z-BASIC
`A$(I)=X$`	`MID$(A$,I,1)=X$`
`A$(I,J)=X$`	`MID$(A$,I,J-I+1)=X$`

Converting Programs to Z-BASIC

Multiple Assignments

Some BASICs allow a statement of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. Z-BASIC would interpret the second equal sign as a logical operator and set B equal to minus one (−1) if C equaled zero. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

Multiple Statements

Some BASICs use a backslash (\) to separate multiple statements on a line. With Z-BASIC, be sure all statements on a line are separated by a colon (:).

Mat Functions

Programs using the MAT functions available in some BASICs must be re-written using FOR...NEXT loops to execute properly.

APPENDIX B
Converting Programs to Z-BASIC

NEW FEATURES IN Z-BASIC, RELEASE 1.00

The execution of BASIC programs written under previously released versions of BASIC, may be affected by some of the new features in Z-BASIC. Before attempting to run such programs, check for the following:

1. New reserved words: CALL, CHAIN, COMMON, WHILE, WEND, WRITE, OPTION BASE, RANDOMIZE, COM, KEY, LOCATE, BEEP, DATE$, and TIME$.

2. Conversion from floating point to integer values results in rounding, as opposed to truncation. This affects not only assignment statements (e.g., I%=2.5 results in I%=3), but also affects function and statement evaluations (e.g., TAB(4.5) goes to the 5th position, A(1.5) yields A(2), and X=11.5 MOD 4 yields 0 for X.)

3. The body of a FOR...NEXT loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

4. Division by zero and overflow no longer produce fatal errors.

5. The RND function has been changed so that RND with no argument is the same as RND with a positive argument. The RND function generates the same sequence of random numbers with each RUN, unless RANDOMIZE is used.

6. The rules for printing single-precision and double-precision numbers have been changed.

7. String space is allocated dynamically, and the first argument in a two-argument CLEAR statement sets the end of memory. The second argument sets the amount of stack space.

APPENDIX B

Converting Programs to Z-BASIC

8. Responding to INPUT with too many or too few items, or with non-numeric characters instead of digits, causes the message "?Redo from start" to be printed. If a single variable is requested, a carriage return may be entered to indicate the default values of 0 for numeric input or null for string input.

   However, if more than one variable is requested, entering a carriage return will cause the "?Redo from start" message to be printed because too few items were entered. No assignment of input values is made until an acceptable response is given.

9. There are two new field formatting characters for use with PRINT USING. An ampersand is used for variable length string fields, and an underscore signifies a literal character in a format string.

10. If the expression supplied with the WIDTH statement is 255, BASIC uses an "infinite" line width, that is, it does not insert carriage returns. WIDTH LPRINT may be used to set the line width of the line printer.

11. The at sign (@) and underscore are no longer used as editing characters.

12. Variable names are significant up to 40 characters and can contain embedded reserved words. However, reserved words must now be delimited by spaces. To maintain compatibility with earlier versions of BASIC, spaces will be automatically inserted between adjoining reserved words and variable names. WARNING: This insertion of spaces may cause the end of a line to be truncated if the line length is close to 255 characters.

13. BASIC programs may be saved in a protected binary format.

APPENDIX C

ASCII Character Codes and Graphic Symbols

OCT = Octal; DEC = Decimal; HEX = Hexadecimal; CHAR = The ASCII character (or function) represented by the code; KEY = The key pressed to produce the code; CTRL = The key pressed in conjunction with the CTRL (Control) key to produce the code; DESCRIPTION = A brief description of the character/function; SYMBOL = The graphics character normally produced while in the graphics mode (unless user-defined).

OCT	DEC	HEX	CHAR	KEY	CTRL	DESCRIPTION
000	0	00	NUL	...	@	Null, tape feed.
001	1	01	SOH	...	A	Start of Heading.
002	2	02	STX	...	B	Start of text.
003	3	03	ETX	...	C	End of text.
004	4	04	EOT	...	D	End of transmission.
005	5	05	ENQ	...	E	Enquiry.
006	6	06	ACK	...	F	Acknowledge.
007	7	07	BEL	...	G	Rings Bell.
010	8	08	BS	BACK SPACE	H	Backspace; also FEB, Format Effector Backspace.
011	9	09	HT	TAB	I	Horizontal Tab.
012	10	0A	LF	LINE FEED	J	Line Feed: advances cursor to next line.
013	11	0B	VT	...	K	Vertical tab (VTAB).
014	12	0C	FF	...	L	Form feed to top of next page.
015	13	0D	CR	RETURN	M	Carriage Return to beginning of line.
016	14	0E	SO	...	N	Shift Out.
017	15	0F	SI	...	O	Shift In.
020	16	10	DLE	...	P	Data link escape.
021	17	11	DC1	...	Q	Device control 1: turns transmitter on (XON).
022	18	12	DC2	...	R	Device control 2.
023	19	13	DC3	...	S	Device control 3: turns transmitter off (XOFF).
024	20	14	DC4	...	T	Device control 4.
025	21	15	NAK	...	U	Negative acknowledge: also ERR (error).
026	22	16	SYN	...	V	Synchronous idle (SYNC).

APPENDIX C

ASCII Character Codes and Graphic Symbols

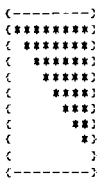
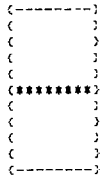
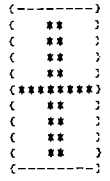
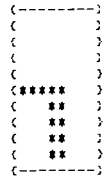
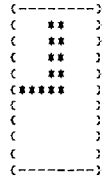
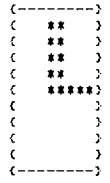
OCT	DEC	HEX	CHAR	KEY	CTRL	DESCRIPTION
027	23	17	ETB	...	W	End of transmission block.
030	24	18	CAN	...	X	Cancel (CANCL). Cancels current escape sequence.
031	25	19	EM	...	Y	End of medium.
032	26	1A	SUB	...	Z	Substitute.
033	27	1B	ESC	ESC	[	Escape.
034	28	1C	FS	...	\	File separator.
035	29	1D	GS	...	]	Group separator.
036	30	1E	RS	...	^	Record separator.
037	31	1F	US	...	_	Unit separator.
040	32	20	SP		...	Space (Spacebar).
041	33	21	!	!	...	Exclamation point.
042	34	22	"	"		Quotation mark.
043	35	23	#	#	...	Number sign.
044	36	24	$	$	...	Dollar sign.
045	37	25	%	%	...	Percent sign.
046	38	26	&	&	...	Ampersand.
047	39	27	'	'	...	Acute accent or apostrophe.
050	40	28	(	(	...	Open parenthesis.
051	41	29	)	)	...	Close parenthesis.
052	42	2A	*	*	...	Asterisk.
053	43	2B	+	+	...	Plus sign.
054	44	2C	,	,	...	Comma.
055	45	2D	-	-	...	Hyphen or minus sign.
056	46	2E	.	.	...	Period.
057	47	2F	/	/	...	Slash.
060	48	30	0	0	...	Number 0.
061	49	31	1	1	...	Number 1.
062	50	32	2	2	...	Number 2.
063	51	33	3	3	...	Number 3.
064	52	34	4	4	...	Number 4.
065	53	35	5	5	...	Number 5.
066	54	36	6	6	...	Number 6.
067	55	37	7	7	...	Number 7.
070	56	38	8	8	...	Number 8.
071	57	39	9	9	...	Number 9.
072	58	3A	:	:	...	Colon.

APPENDIX C

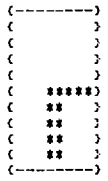
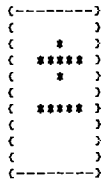
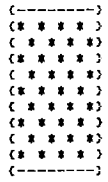
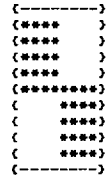
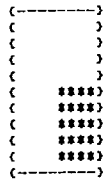
ASCII Character Codes and Graphic Symbols

OCT	DEC	HEX	CHAR	KEY	CTRL	DESCRIPTION	SYMBOL
073	59	3B	;	;	...	Semicolon.	
074	60	3C	<	<	...	Less than.	
075	61	3D	=	=	...	Equal sign.	
076	62	3E	>	>	...	Greater than.	
077	63	3F	?	?	...	Question mark.	
100	64	40	@	@	...	At sign.	
101	65	41	A	A	...	Letter A.	
102	66	42	B	B	...	Letter B.	
103	67	43	C	C	...	Letter C.	
104	68	44	D	D	...	Letter D.	
105	69	45	E	E	...	Letter E.	
106	70	46	F	F	...	Letter F.	
107	71	47	G	G	...	Letter G.	
110	72	48	H	H	...	Letter H.	
111	73	49	I	I	...	Letter I.	
112	74	4A	J	J	...	Letter J.	
113	75	4B	K	K	...	Letter K.	
114	76	4C	L	L	...	Letter L.	
115	77	4D	M	M	...	Letter M.	
116	78	4E	N	N	...	Letter N.	
117	79	4F	O	O	...	Letter O.	
120	80	50	P	P	...	Letter P.	
121	81	51	Q	Q	...	Letter Q.	
122	82	52	R	R	...	Letter R.	
123	83	53	S	S	...	Letter S.	
124	84	54	T	T	...	Letter T.	
125	85	55	U	U	...	Letter U.	
126	86	56	V	V	...	Letter V.	
127	87	57	W	W	...	Letter W.	
130	88	58	X	X	...	Letter X.	
131	89	59	Y	Y	...	Letter Y.	
132	90	5A	Z	Z	...	Letter Z.	
133	91	5B	[	[	...	Open brackets.	
134	92	5C	\	\	...	Reverse slash.	
135	93	5D	]	]	...	Close brackets.	
136	94	5E	^	^	...	Up arrow/caret.	

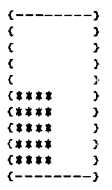
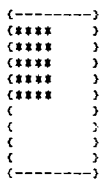
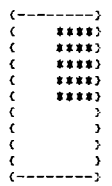
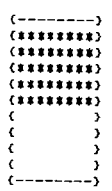
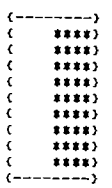
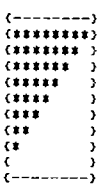
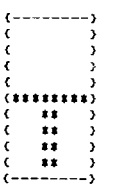
ASCII Character Codes and Graphic Symbols

OCT	DEC	HEX	CHAR	KEY	CTRL	DESCRIPTION	SYMBOL
137	95	5F	_	_	...	Underscore.	
140	96	60	`	`	...	Grave accent.	
141	97	61	a	a	...	Letter a.	
142	98	62	b	b	...	Letter b.	
143	99	63	c	c	...	Letter c.	
144	100	64	d	d	...	Letter d.	
145	101	65	e	e	...	Letter e.	

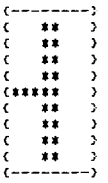
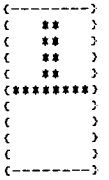
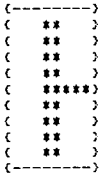
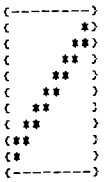
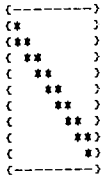
ASCII Character Codes and Graphic Symbols

OCT	DEC	HEX	CHAR	KEY	CTRL	DESCRIPTION	SYMBOL
146	102	66	f	f	...	Letter f.	
147	103	67	g	g	...	Letter g.	
150	104	68	h	h	...	Letter h.	
151	105	69	i	i	...	Letter i.	
152	106	6A	j	j	...	Letter j.	
153	107	6B	k	k	...	Letter k.	
154	108	6C	l	l	...	Letter l.	

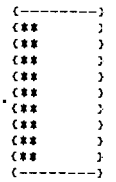
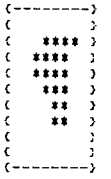
ASCII Character Codes and Graphic Symbols

OCT	DEC	HEX	CHAR	KEY	CTRL	DESCRIPTION	SYMBOL
155	109	6D	m	m	...	Letter m.	
156	110	6E	n	n	...	Letter n.	
157	111	6F	o	o	...	Letter o.	
160	112	70	p	p	...	Letter p.	
161	113	71	q	q	...	Letter q.	
162	114	72	r	r	...	Letter r.	
163	115	73	s	s	...	Letter s.	

ASCII Character Codes and Graphic Symbols

OCT	DEC	HEX	CHAR	KEY	CTRL	DESCRIPTION	SYMBOL
164	116	74	t	t	...	Letter t.	
165	117	75	u	u	...	Letter u.	
166	118	76	v	v	...	Letter v.	
167	119	77	w	w	...	Letter w.	
170	120	78	x	x	...	Letter x.	
171	121	79	y	y	...	Letter y.	
172	122	7A	z	z	...	Letter z.	

ASCII Character Codes and Graphic Symbols

OCT	DEC	HEX	CHAR	KEY	CTRL	DESCRIPTION	SYMBOL
173	123	7B	{	{	...	Left brace.	
174	124	7C	¦	¦	...	Vertical bar (broken).	
175	125	7D	}	}	...	Right brace.	
176	126	7E	~	~	...	Tilde.	
177	127	7F	DEL	DELETE	...	Delete (rubout).	
200-377	128-255	80-FF				unassigned.	

APPENDIX D

Mathematical Functions

DERIVED FUNCTIONS

Functions that are not intrinsic to BASIC may be calculated as follows.

Function	BASIC Equivalent
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X)=-ATN(X/SQR(-X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X)=ATN(SQR(X*X-1)) -SGN(SGN(X)-1)*SGN(X)*3.1416
INVERSE COSECANT	ARCCSC(X)=ATN(1/SQR(X*X-1) -SGN(SGN(X)-1)*SGN(X)*3.1416
INVERSE COTANGENT	ARCCOT(X)=1.5708-ATN(X)
HYPERBOLIC SINE	SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X)=(EXP(X)-EXP(-X))/(EXP(X)+EXP(-X))
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X)=(EXP(X)+EXP(-X))/(EXP(X)-EXP(-X))
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X)
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1))/2

APPENDIX E

Assembly Language Subroutines

BRIEF

All versions of Zenith BASIC have provisions for interfacing with assembly language subroutines via the USR function and the CALL statement.

Following is a detailed discussion of assembly language interface, memory allocation and stack space.

Details

The USR function allows assembly language subroutines to be called in the same way BASIC Intrinsic functions are called. However, the CALL statement is the recommended way of interfacing 8086 machine language programs with BASIC. It is compatible with more languages than is the USR function call, it produces more readable source code, and it can pass multiple arguments.

MEMORY ALLOCATION

Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s) with the \M: switch.

In addition to the BASIC interpreter code area, Z-BASIC uses up to 64K of memory beginning at its data (DS) segment.

If, when an assembly language subroutine is called, more stack space is needed, BASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC's stack must be restored, however, before returning from the subroutine.

APPENDIX E

Assembly Language Subroutines

The assembly language subroutine may be loaded into memory by means of the operating system or the BASIC POKE statement. If the user has the Zenith Utility Software Package, the routines may be assembled with the MACRO-86 assembler and linked using the MS-LINK Linker, but not loaded. To load the program file, the user should observe these guidelines:

1. The subroutines must not contain any long references.

2. Skip over the first 512 bytes of the MS-LINK output file, then read in the rest of the file.

CALL Statement

As we mentioned earlier, the CALL statement is the recommended way of interfacing 8086 machine language programs with BASIC. It is further suggested that the old style user-call USR(n) not be used.

Format: `CALL <variable name> [(<argument list>)]`

<variable name> contains the segment offset that is the starting point in memory of the subroutine being CALLed.

<argument list> contains the variables or constants, separated by commas, that are to be passed to the routine.

The CALL statement conforms to the INTEL PL/M-86 calling conventions outlined in Chapter 9 of the INTEL PL/M-86 Compiler Operator's Manual. BASIC follows the rules described for the MEDIUM case (summarized in the following discussion).

Invoking CALL Statement

Invoking the CALL statement causes the following to occur:

1. For each parameter in the argument list, the two-byte offset of the parameter's location within the data segment (DS) is pushed onto the stack.

2. BASIC's return address code segment (CS), and offset (IP) are pushed onto the Stack.

3. Control is transferred to the user's routine via an 8086 long call to the segment address given in the last DEF SEG statement and the offset given in <variable name>.

These actions are illustrated by the two following diagrams, which illustrate first, the state of the stack at the time of the CALL statement, and second, the condition of the stack during execution of the called subroutine.

Stack Diagram 1

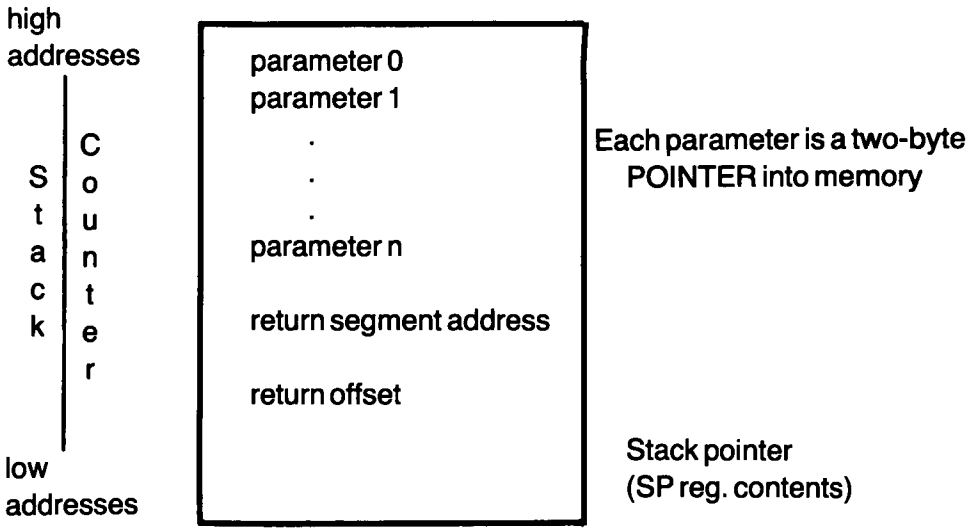


Diagram 1

Stack Layout when CALL Statement Is activated

The user's routine now has control. Parameters may be referenced by moving the stack pointer (SP) to the base pointer (BP) and adding a positive offset to (BP).

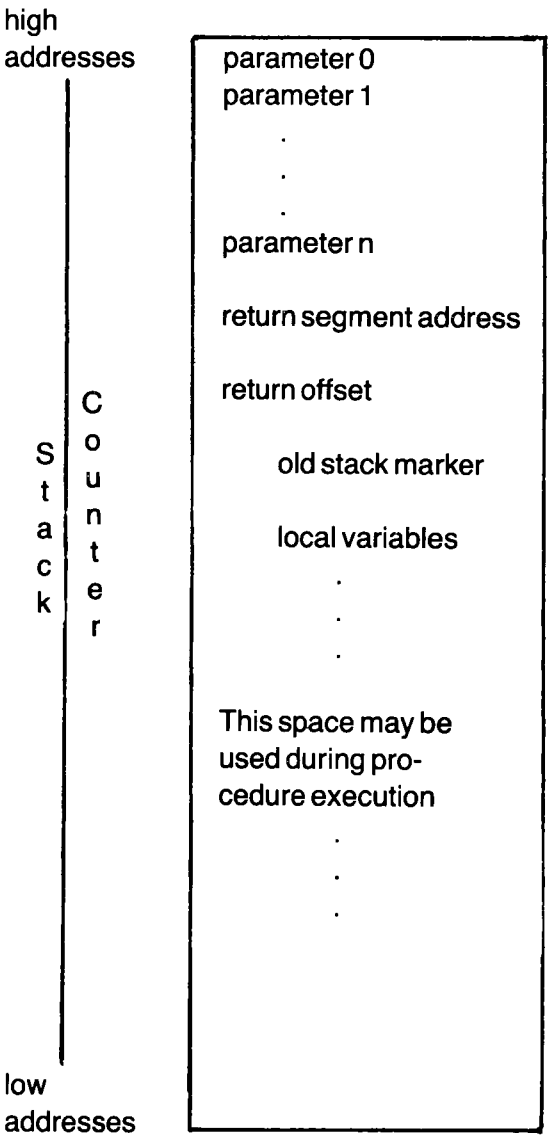


Diagram 2

Stack Layout During Execution of a CALL statement

Diagram 2

APPENDIX E
Assembly Language Subroutines

Coding Rules

You must observe the following rules when coding a subroutine:

1. The called routine may destroy the SX, BX, CX, DX, SI, DE, and BP registers.

2. The called program MUST know the number and length of the parameters passed. References to parameters are positive offsets added to (BP) (assuming the called routine moved the current stack pointer into BPI i.e., MOV BP,SP). That is, the location of P1 is at 8(BP), p2 is at 6(BP), p3 is at 4(BP),...etc.

3. The called routine must do a RET <n> (where <n> is two times the number of parameters in the argument list) to adjust the stack to the start of the calling sequence.

4. Values are returned to BASIC by including in the argument list the variable name(s) which will receive the result.

5. If the argument is a string, the parameter's offset points to three-bytes called the "String Descriptor." Byte zero of the string descriptor contains the length of the string (0 to 255). Bytes one and two, respectively, are the lower and upper eight-bits of the string starting address in string space.

NOTE: If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +" " to the string literal in the program.

Example:

```
20 A$ = "BASIC"+""
```

This will force the string literal to be copied into string space. Now the string may be modified without affecting the program.

6. Strings may be altered by user routines, but the length *must not* be changed. BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

Example:

Assemble the subroutine.

```
A:MASM CALL,CALL,CALL;
The Microsoft MACRO Assembler
Version 1.06, Copyright (C) Microsoft Inc. 1981,82

Warning Severe
Errors  Errors
0       0
```

Link the subroutine.

```
A:LINK CALL:

   Microsoft Object Linker V1.10
(C) Copyright 1981 by Microsoft Inc.

Warning: No STACK segment

There was 1 error detected.
```

(NOTE: This error is ok. The subroutine does not contain a stack since it uses Z-BASIC's.)

Convert the subroutine to binary code.

```
A:EXE2BIN CALL
Exe2bin version 1.5
```

Assembly Language Subroutines

This is a listing of the subroutine generated by MASM.

A:TYPE CALL.LST

```
The Microsoft MACRO Assembler                    08-20-82          PAGE     1-1

                        PAGE ,132
0000
                        FUNC    SEGMENT

                                ASSUME  CS:FUNC

0000                    START   PROC    FAR
0000  8B EC                      MOV     BP,SP                   ;Set up frame pointer
0002  8B 76 06                   MOV     SI,6[BP]                ;SI = pointer to paraml
0005  8B 04                      MOV     AX,WORD PTR [SI]        ;AX = integer value
0007  8B 76 04                   MOV     SI,4[BP]                ;SI = pointer to param2
000A  03 C0                      ADD     AX,AX                   ;AX = AX * 2
000C  89 04                      MOV     WORD PTR [SI], AX       ;Save it
000E  CA 0004                    RET     4

0011                    START   ENDP
0011                    FUNC    ENDS
                                END     START


The Microsoft MACRO Assembler                    08-20-82          PAGE     Symbols-1

Segments and groups:

                N a m e                 Size     align     combine class

FUNC . . . . . .                        0011     PARA      NONE

Symbols:

                N a m e                 Type     Value     Attr

START. . . . . .                        F PROC   0000      FUNC     Length =0011

Warning Severe
Errors  Errors
0       0
```

When calling Z-BASIC, set the /M switch to 32768:

ZBASIC /M:32768

The following is the BASIC program. The value in line 10 is for a 192K Z-100. For a 128K machine, make the value &H1F00.

```
10 DEF SEG = &H2F00     'set base of Call/Peak/Poke to 2F00:0000
20 GOSUB 80                          'load program
30 Y%=5                               'set Y%
40 MULT = &H0                         'set address of program
50 CALL MULT(Y%,X%)                  'call routine
60 PRINT X%                          'print result
70 END                               'done
80 OPEN "R",1,"CALL.BIN",2           'Open binary file
90 FIELD #1, 2 AS A$                  'set 2-byte field
100 FOR X=&H0 TO (LOF(1)+1) STEP 2 'for next to read every byte
110 GET #1,X/2+1                      'get next pair of bytes
120 Q%=CVI(A$)                        'convert to 16-bit integer
130 M%=Q% MOD 256                     'split into 8 high and
140 L%=INT (Q%/256)                   '  8 low-bits
150 POKE X,M% AND &HFF               'poke data into memory
160 POKE X+1,L% AND &HFF              '  locations
170 NEXT X                           'get next pair
180 RETURN
```

APPENDIX E

Assembly Language Subroutines

USR FUNCTION CALLS

Although the CALL statement is the recommended way of calling assembly language subroutines, the USR function call is still available for compatibility with previously-written programs.

Format

The format of the USR function call is:

```
USR[<digit>][(argument)]
```

<***digit***> is from 0 to 9. <digit> specifies which USR routine is being called. If <digit> is omitted, USR0 is assumed.

(***argument***) is any numeric or string expression. Arguments are discussed in detail below.

In this implementation of BASIC, a DEF SEG statement ***must*** be executed prior to a USR call to assure that the code segment points to the subroutine being called. The segment address given in the DEF SEG statement determines the starting segment of the subroutine.

For each USR function, a corresponding DEF USR statement must have been executed to define the USR call offset. This offset and the currently active DEF SEG address determine the starting address of the subroutine.

When the USR function call is made, register [AL] contains a value which specifies the type of argument that was given. The value in [AL] may be one of the following:

2	Two-byte integer (two's complement)
3	String
4	Single-precision floating point number
8	Double-precision floating point number

APPENDIX E

Assembly Language Subroutines

If the argument is a number, the [BX] register pair points to the floating point accumulator (FAC) where the argument is stored:

FAC — is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

FAC-1 — contains the highest seven bits of mantissa with leading 1 suppressed (implied). Bit seven is the sign of the number (0=positive, 1=negative).

If the argument is n integer:

FAC-2 — contains the upper eight bits of the argument.

FAC-3 — contains the lower eight bits of the argument.

If the argument is a single-precision floating point number:

FAC-2 — contains the middle eight bits of mantissa.

FAC-3 — contains the lowest eight bits of mantissa.

If the argument is a double-precision floating point number:

FAC-7 to FAC-4 — contain four more bytes of mantissa (FAC-7 contains the lowest eight bits).

If the argument is a string:

the [DX] register pair points to three-bytes called the "string descriptor." Byte zero of the string descriptor contains the length of the string (0 to 255). Bytes one and two, respectively, are the lower and upper eight bits of the string starting address in BASIC's data segment.

NOTE: If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. See the CALL statement above.

Usually, the value returned by a USR function is the same type (integer, string, single-precision, or double-precision) as the argument that was passed to it.

APPENDIX F

Communication I/O

Since the communication port is opened as a file, all Input/Output statements that are valid for disk files are valid for COM.

COM sequential input statements are the same as those for disk files. They are: INPUT#<file name>, LINE INPUT #<file number>, and the INPUTS function.

COM sequential output statements are the same as those for disk, and are: PRINT #<file number>, and PRINT #<file number> USING.

Refer to INPUT and PRINT sections for details of coding syntax and usage.

GET and PUT are only slightly different for COM files, see The GET and PUT statements for COM files.

THE COM I/O FUNCTIONS

The most difficult aspect of asynchronous communication is being able to process characters as fast as they are received. At rates above 2400 bps, it is necessary to suspend character transmission from the host long enough to "catch up". On some systems, this can be done by sending XOFF (**CTRL-S**) to the host and XON (**CTRL-Q**) when ready to resume. (Be sure to obtain this information in the case you need to use this method.)

BASIC provides three functions which help in determining when an "over-run" condition is eminent. These are:

LOC(x) Returns the number of characters in the input queue waiting to be read. The input queue can hold 120 characters. If there are more than 120 characters in the queue, LOC(X) returns 120. Since a string is limited to 255 characters, this practical limit alleviates the need for the programmer to test for string size before reading data into it. If fewer than 120 characters remain in the queue, LOC(X) returns the actual count.

APPENDIX F

Communication I/O

LOF(x) Returns the amount of free space in the input queue. That is, 120 − LOC(x). Use of LOF may be used to detect when the input queue is getting full. In practicality, LOC is adequate for this purpose as will be demonstrated in the programming example.

EOF(x) If true (−1), indicates Z (1AH) has been received. Returns false (0) Z has not been received. If there are no characters in the inut queue, then the system will wait until a character is received.

Possible Errors:

1. `Communication Buffer Overflow` If a read is attempted after the input queue is full, (i.e. LOF(x) returns 0).

2. `Device I/O Error` If any of the following line conditions are detected on receive; Overrun Error (OE), Framing Error (FE), or Break Interrupt (BI). The error is reset by subsequent inputs but the character causing the error is lost.

This error message will also be returned if the input queue holds less than the number of characters requested by the INPUT$ function. To avoid this condition, use the example shown in the following discussion, or poll the input queue for the number of characters with the LOC(x) function.

```
10 OPEN "COM1:1200,N,8,2" AS #1  :REM OPEN COM1: CHANNEL
20 GOSUB 100 : PRINT A$          :REM READ 10 CHARACTERS FROM COM1: BUFFER
30 GOTO 20                       :REM GO INTO A LOOP
100 IF LOC(1)<10 THEN 100        :REM WAIT FOR 10 CHARACTERS IN BUFFER
110 A$=INPUTS(10,#1)             :REM READ 10 CHARACTERS
120 RETURN
```

3. `Device Fault` If Data Set Ready (DSR) is lost during I/O.

APPENDIX F
Communication I/O

THE INPUT$ FUNCTION FOR COM FILES

The INPUT$ function is preferred over the INPUT and LINE INPUT statements when reading COM files, since all ASCII characters may be significant in communications. INPUT is least desirable because input stops when a comma (,) or RETURN is seen and LINE INPUT terminates when a RETURN is seen.

INPUT$ allows all characters read to be assigned to a string. Recall from the rules for coding that INPUT$ will return X characters from the #Y file. The following statements then are most efficient for reading a COM file:

```
10 WHILE LOC(1)<>0
20 A$=INPUT$(LOC(1),#1)
30 ...
40 ... Process data returned in A$...
50 ...
60 WEND
```

The previous sequence of statements read: ".. While there is something in the input queue, return the number of characters in the queue and store them in A$. Continue as long as there are characters present in the input queue.

The GET and PUT Statements for COM Files

Format: `GET<file number>,<nbytes>`
`PUT<file number>,<nbytes>`

Function: GET and PUT allow fixed length I/O for COM.

<file number> Is an integer expression returning a valid file number.

<nbytes> Is an integer expression returning the number of bytes to be transferred into or out of the file buffer. nbytes cannot exceed 120.

Because of the low performance associated with telephone line communication, it is recommended that GET and PUT not be used in such applications.

APPENDIX F

Communication I/O

Examples:

The following program enables the Z-100 computer to be used as a conventional terminal. Besides full duplex communication with a host, the TTY program allows ASCII text to be "down-loaded" to a file. Conversely, a file may be "up-loaded" (transmitted) to another machine.

In addition to demonstrating the elements of asynchronous communication, this program should be useful in transferring BASIC programs (Saved with the A option) and ASCII text to and from the Z-100.

NOTE: This program is set up to communicate with Microsoft's DEC-20, that is, the use of XON and XOFF. You may want to further modify it for your environment.

The TTY Program (An exercise in communication I/O).

```
10 SCREEN 0,0
15 KEY OFF:CLS:CLOSE
20 DEFINT A-Z
25 LOCATE 25,1
30 PRINT STRING$(60,"")
40 FALSE=0:TRUE= NOT FALSE
50 MENU=5 '    When CTRL-E is hit, the menu is displayed
60 XOFF$=CHR$(19) :XON$=CHR$(17)
100 LOCATE 25,1:PRINT "Async TTY Program, Press CTRL-E to display menu";
105 LOCATE 1,1:PRINT "Async TTY Program"
110 LINE INPUT "Speed? ";SPEED$
120 COMFIL$="COM1:"+SPEED$+",N,8"
130 OPEN COMFIL$ AS #1
140 OPEN "SCRN:" FOR OUTPUT AS #2
200 PAUSE=FALSE
210 A$=INKEY$: IF A$="" THEN 230
220 IF ASC(A$)=MENU THEN 300 ELSE PRINT #1,A$;
230 IF LOC(1)=0 THEN 210
240 IF LOC(1)>82 THEN PAUSE=TRUE: PRINT #1,XOFF$;
250 A$=INPUT$(LOC(1),#1)
260 PRINT #2,A$;:IF LOC(1)>0 THEN 240
270 IF PAUSE THEN PAUSE=FALSE:PRINT #1,XON$;
280 GOTO 210
300 LOCATE 1,1:PRINT STRING$(30," ") :LOCATE 1,1
310 LINE INPUT"FILE? ";DSKFIL$
400 LOCATE 1,1:PRINT STRING$(30," "):LOCATE 1,1
410 LINE INPUT"(T)ransmit (R)eceive, or (E)xit? ";TXRX$
415 IF (TXRX*<>"T") AND (TXRX$<>"R") AND (TXRX$<>"E" THEN 400
```

APPENDIX F
Communication I/O

```
417 IF TXRX$="E" THEN 9999
420 IF TXRX$="T" THEN OPEN DSKFIL$ FOR INPUT AS #3:GOTO 1000
430 OPEN DSKFIL$ FOR OUTPUT AS #3
440 PRINT #1,CHR$(13) ;
500 IF LOC(1)=0 THEN GOSUB 600
510 IF LOC(1)>82 THEN PAUSE=TRUE: PRINT #1,XOFF$;
520 A$=INPUT$(LOC(1),#1)
530 PRINT #3,A$;:IF LOC(1)>0 THEN 510
540 IF PAUSE THEN PAUSE=FALSE:PRINT #1,XON$;
550 GOTO 500
600 FOR I=1 TO 5000
610 IF LOC(1)<>0 THEN I=9999
620 NEXT I
630 IF I>9999 THEN RETURN
640 CLOSE #3:CLS:LOCATE 25,10: PRINT "* Download complete*";
650 GOTO 200
1000 WHILE NOT EOF(3)
1010 A$=INPUT$(1,#3)
1020 PRINT #1,A$;
1030 WEND
1040 PRINT #1,CHR$(26);  'CTRL-Z to make close file.
1050 CLOSE #3:CLS:LOCATE 25,10:PRINT "** Upload complete **
";
1060 GOTO 200
9999 CLOSE:KEY ON
```

NOTES ON THE TTY PROGRAMMING EXAMPLE:

Line No.	Comments
10	Turns off the graphics mode and clears the reverse video mode (returns to normal display).
15	Turns off the soft key display, clears the screen, and makes sure that all files are closed.

Asynchronous implies character I/O as opposed to line or block I/O. Therefore, all prints (either to the COM file or screen) are terminated with a semicolon (;). This retards the RETURN line-feed normally issued at the end of a PRINT statement.

20	Define all numeric variables as INTEGER.
25-30	Clears the 25th line starting at column 1.

APPENDIX F

Communication I/O

Line No.	Comments
40	Define Boolean TRUE and FALSE.
50	Defines the value of the control key (CTRL-E) that will display the MENU.
60	Defines the value for the XON and XOFF characters (11H, 17 Dec and 13H, 19 Dec, respectively).
100-130	Prints program-ID and asks for baud rate (speed). Opens communications to file number one, no parity, eight data bits.
200-280	This section performs full-duplex I/O between the video screen and the device connected to the RS-232 connector as follows: 1. Read a character from the keyboard into A$. Note that INKEY$ returns a null string if no character is waiting. 2. If no character is waiting then go see if any characters are being received. If a character is waiting at the keyboard then: 3. If the character was the MENU Key, then the user is ready to download a file, so go get file name. 4. If character (A$) is not the MENU key then send it by writing to the communication file (PRINT #1...). 5. At 230 see if any characters are waiting in COM buffer. If not, then go back and check keyboard. 6. At 240, if more than 82 characters are waiting then, set PAUSE flag saying we are suspending input and send XOFF to host stopping further transmission.

APPENDIX F

Communication I/O

Line No.	Comments
	7. At 250-260, read and display contents of COM buffer on screen until empty. Continue to monitor size of COM buffer (in 240). Suspend transmission if we fall behind.
	8. Finally, resume hose transmission by sending XON only if suspended by previous XOFF. Repeat process until MENU Key struck.
300-310	Get disk file name we are down-loading to.
400-430	Asks if file named is to be transmitted (up-load) or received (down-loaded). Open the file as number 3.
440	Sends a RETURN to the host to begin the down-load. This program assumes that the last command sent to the host was to begin such a transfer and was missing only the terminating RETURN. If a DEC System is the host, then such a command might be:
	COPY TTY:=MANUAL.MEM<CTRL-E>
WHERE:	The MENU Key (CTRL-E) was struck instead of RETURN.
500	When no more characters are being received (LOC(x) returns 0), then perform a timeout routine (explained later).
510	Again, if more than 82 characters are waiting, signal a pause and send XOFF to the host while we catch-up.
520-530	Read all characters in COM queue (LOC(x)) and write them to disk (PRINT #3..) until we are caught up.
540-550	If a pause was issued, restart host by sending XON and clear the pause flag. Continue process until no characters are received for a predetermined time.

APPENDIX F

Communication I/O

Line No.	Comments
600-650	This is the time-out subroutine. The FOR loop count was determined by experimentation. In short, if no character is received from the host for 17-20 seconds, then transmission is assumed complete. If any character is received during this time (line 610) then set I well above FOR loop range to exit loop and then return to caller. If host transmission is complete, close the disk file and return to being a terminal.
1000-1060	Transmit routine. Until end of disk file do: Read one character into A$ with INPUT$ statement. Send character to COM device in 1020. Send a CTRL-Z at end of file in 1040 to close the receiving devices file. Finally, in lines 1050 and 1060, close our disk file, print completion message and go back to conversation mode in line 200.
9999	This line closes the COM file left open and restores the soft key display.

EVENT TRAPPING

The following are defined as "event specifiers":

COM (n) where n is the number of the COM channel (one or two)

KEY (n) where n is a function KEY Number (one-12). One through 12 are the Soft Keys F1 through F12.

We add the following statements:

```
ON <event specifier> GOSUB <line number>
```

This sets up an event trap line number for the specified event. A <line number> of 0 disables trapping for this event.

```
<event specifier> ON
<event specifier> OFF
<event specifier> STOP
```

These statements control the activation/deactivation of event trapping. When an event is ON, if a non-zero line number is specified for the trap with an ON statement then everytime BASIC starts a new statement it will check to see if the specified event has occurred (a function key was struck, a com character has come in) and if so, it will perform a GOSUB to the line specified in the ON statement.

When an event is OFF, no trapping takes place and the event is not remembered even if it takes place.

When an event is "stopped" (it must be turned on first) no trapping can take place, but if the event happens this is remembered so an immediate trap will take place when an <event> ON is executed.

When a trap is made for a particular event the trap automatically causes a "stop" on that event so recursive traps can never take place the "return" from the trap routine automatically does an ON unless an explicit OFF has been performed inside the trap routine.

When an error trap takes place this automatically disables all trapping.

Trapping will never take place when BASIC is not executing a program.

Special notes about each type of trap:

KEY Trapping.

No type of trapping is activated when BASIC in direct mode. In particular, function keys resume their standard expansion meaning during input.

A key that causes a trap is not available for examination with the INPUT or INKEY$ statements so the trap routine for each key must be different if a different function is desired.

COM Trapping.

Typically the COM trap routine will read an entire message from the COM port before returning back. It is not recommended to use the COM trap for single character messages since at high baud rates the overhead of trapping and reading for each individual character may allow the interrupt buffer for COM to overflow.

Here is an example of event trapping using the F1 key:

```
10 KEY(1)ON
20 ON KEY(1) GOSUB 100
30 GOTO 30
.
.
.
100 BEEP: KEY(1)OFF : RETURN
```

The program will turn on the event trapping and cycle in line 30 until you press the F1 key. At that point, the program will execute line 20 and go to the subroutine in line 100 where it will sound the tone, turn the key event off and return from the subroutine. If you press the F1 key a second time, nothing will happen because the event trapping has been turned off.

APPENDIX G

Glossary

A GLOSSARY OF COMMONLY USED COMPUTER TERMS

Acoustic coupler (Modem) - One of the two types of modems: a device you can connect between a standard telephone handset and a Computer to communicate with other Computers. A modem will translate the normal digital signals of the Computer into tones (and back again) that are transmitted over standard telephone lines. By using an acoustic coupler modem, you can use any telephone with a standard handset on a temporary basis and avoid a permanent connection to the telephone lines. See "Modem" and "Direct-Connect Modem."

Acronym - A word formed from letters found in a name, term, or phrase. For example, FORTRAN is formed from the words FORmula TRANslator.

Address - The label, name, or number identifying a register, location, or unit where data is stored. In most cases, address refers to a location in Computer memory.

Algorithm- A defined set of instructions that will lead to the logical conclusion of a task.

Alpha- The letters of the English alphabet.

Alphanumeric - Letters, numbers, punctuation, and symbols used to represent information or data.

ALU - Arithmetic Logic Unit. This section of the Computer performs the arithmetic, logical, and comparative functions of an operation.

ANSI - American National Standards Institute. This organization publishes standards used by many industries, including the Computer industry. Most noted are those standards established for Computer languages such as FORTRAN and COBOL.

Analyst - A person who has been trained to define problems and develop solutions. In the Computer industry, an analyst will also develop algorithms for Computer programs.

APPENDIX G

Glossary

Application - A system, problem, or task to which a Computer has been assigned.

Application program - A program or set of programs designed to accomplish a specific task like word processing.

Argument - A term used to describe a value in a variable, statement, command, or element of an array or matrix table.

Array - A series of items arranged in a pattern. In computing, this term is used to describe a table with one or more dimensions.

Artificial intelligence- A term used to describe the capability of a machine that can perform functions normally associated with human intelligence: reasoning, creativity, and self-improvement.

ASCII - American Standard Code for Information Interchange, a code used by most Computers, including those sold by Heath and Zenith. It is the industry standard used to transmit infcrmation to printers, other Computers, and other peripheral devices. The most notable exception is some of the IBM equipment which uses an EBCDIC code. See "EBCDIC."

Assemble - To prepare a machine usable code from a symbolic code.

Assembler- A Computer program used to assemble machine code from symbolic code.

Assembly language - A Computer programming language that is heavily machine oriented and makes use of mnemonics for instructions, operands, and pseudo-operations.

Asynchronous - A mode of operation where the next command is started and stopped by special signals. In communication, the signals are referred to as start and stop bits.

Backup - 1. A copy preserved as a protection from the destruction of the original (or processed) data and/or programs. 2. The process of producing a backup.

APPENDIX G

Glossary

BASIC - Beginner's All-purpose Symbolic Instruction Code. An easily learned programming language consisting largely of English words and terms.

Batch processing- An operation where a large amount of similar data is processed by a Computer with little or no operator supervision. See "Interactive Processing."

Baud rate - The rate at which information is transmitted serially from a Computer. Expressed in bits per second.

BCD - Binary Coded Decimal. The method of encoding four bits of Computer memory into a binary representation of one decimal digit (number).

Binary - A numbering system based on two's rather than ten's (decimal). The individual element (or digit) can have a value of zero or one and in Computer memory is known as a bit.

Bit - 1. A single binary element or digit. 2. The smallest element in Computer storage capability.

Bit density - A measure of the number of bits recorded in a given area.

Block diagram - 1. A graphic representation of the logical flow of operations in a Computer program, usually more general than a flow chart. 2. A graphic representation of the hardware configuration of a Computer system.

Boolean algebra - A symbolic system (algebra) named after its developer, George Boole. It is concerned with Computer and binary processes and includes logical operators.

Boot - The process of initializing (or loading) a Computer operating system. Also referred to as "Booting Up."

Bootstrap - A program used by a Computer to initialize (or load) the operating system of the Computer.

Branch - To depart from the sequential flow of an operation as the result of a decision.

APPENDIX G

Glossary

Break - The process of interrupting and (temporarily) halting a sequence of operations, as in a Computer program.

Buffer - An auxiliary storage area for data. Many peripherals have buffers which are used to temporarily store data which the peripheral will use as time permits.

Bug - A term that is widely used to describe the cause of a Computer misoperation. The "bug" may be either in the hardware design or in the software (programs) used by the Computer.

Bus - A circuit (line) used to carry data or power between two or more sources. The S-100 bus, which is used in the Z-100 series Desktop Computer, is composed of one hundred separate bus lines.

Byte - A term used to describe a number of consecutive bits. In microComputers, a byte refers to eight bits and is used to represent one ASCII or EBCDIC character.

Cable - An assembly of one or more conductors used to transmit power or data from a source to a destination and, in some cases, vice-versa.

Character - A letter, number, punctuation, operation symbol, or any other single symbol that a Computer may read, store, or process.

Check (sum) - A method of checking the accuracy of characters transmitted, manipulated, or stored. The check sum is the result of the summation of all the digits involved.

Chip - The term applied to an integrated circuit that contains many electronic circuits. It is sometimes called an IC or an IC chip and sometimes refers to the entire integrated circuit package.

Circuit - A system of electronic elements and connections through which current flows.

COBOL- COmmon Business Oriented Language. This common high-level language is used in a wide number of operations, most notably those dealing with financial transactions.

APPENDIX G

Glossary

Code- A method of representing data in some form, as in an ASCII or EBCDIC form.

Column - A character position in a side-by-side relationship as opposed to a row position which is one above another.

Command - A portion of code that represents an instruction for the Computer.

Communication - The process of transferring information from one point to another.

Compile - The process of producing machine code or pseudo-operational code from a higher-level code, or language, such as COBOL or FORTRAN.

Compiler- The program that compiles machine code from a higher-level code. See "Compile."

Computer - A machine capable of accepting information, processing it by following a set of instructions, and supplying the results of this process.

CP/M - Control Program for Microcomputers. This is a disk-based operating system commonly used by many microcomputers. CP/M is a registered trade mark of Digital Research, Inc.

CPS - Characters Per Second. This term is sometimes used in relating transmission speed, and is more commonly used in rating a printer's instantaneous printing speed.

CPU- Central Processing Unit. The CPU is the brain of a Computer. It is the circuitry which actually processes the information and controls the storage, movement, and manipulation of that data. The CPU contains the ALU and a number of registers for this purpose.

Crash - A term that refers to a Computer or peripheral failure.

CRT - Cathode Ray Tube. This term is used interchangeably with display, screen, and video monitor. It refers to the television-like screen in a Computer or terminal.

APPENDIX G

Glossary

Cursor - A character, usually an underline or graphics block, used to indicate position on a display screen.

Cylinder - Used to describe the tracks in disk units with multiple read-write heads, which can be accessed without mechanical movement of the heads.

Daisy wheel printer - A hard copy device that produces images on paper when a hammer strikes an arm or projection of the print wheel, which looks somewhat like a daisy. The print quality from such printers is usually quite high, similar to that of a quality office electric typewriter.

Data - The general term used to describe information that can be processed by a Computer. Although the term is plural, it is commonly used in a singular form to denote a group of datum.

Data base - A large file of information that is produced, updated, and manipulated by one or more programs.

Data processing - This term usually refers to the act of processing raw data, as by the use of a Computer.

Debug - The process of locating and removing any "bugs" in a Computer system; usually as it applies to software.

Decimal - The numbering system based on ten and comprising the digits 0 through 9.

Delete - To remove or eliminate.

Density - The closeness of space distribution on a storage medium such as a diskette.

Device - A separate mechanical or electronic unit, such as a printer, disk drive, terminal, and so on.

Digit - A single element or sign used to convey the idea of quantity, either by itself or with other numbers of its series.

Digital computer - A Computer in which numbers are used to express data and instructions.

APPENDIX G
Glossary

Direct-connect modem - One of the two types of modems; a device you can connect between a telephone line and a Computer to communicate with other Computers. A modem will translate the normal digital signals of the Computer into tones (and back again) that are transmitted over standard telephone lines. By using a direct-connect modem, you avoid problems associated with high levels of noise and make a more permanent connection to the telephone lines. See "Modem" and "Acoustic Coupler."

Directory - A disk file, listing all of the other files on the diskette and pertinent information about each file.

Disk - A circular metal plate coated with magnetic material and used to store large amounts of data. Also called a hard disk. See "Diskette."

Disk drive - A device used to read data from and to write data onto diskettes.

Diskette - A thin, flexible plastic platter, coated with magnetic material and enclosed in a plastic jacket. It is used to store data and comes in two standard sizes: 5-1/4" and 8" in diameter. Also called a "floppy disk," "floppy diskette," "flexible disk," or "flexible diskette."

Disk operating system - See "DOS."

Display - The television-like screen used by the Computer to present information to the operator.

DOS - Disk operating system - A program or programs that provide basic utility operations and control of a disk based Computer system.

Dot-matrix printer - A hard copy printer that works by forming the printed character through the selection of wires which strike the paper.

Double density - This term is most often applied to the storage characteristics of diskettes, and generally refers to the density of the storage of bits on the diskette surface on each track. It also refers to the density of the diskette tracks, though this is not the common usage.

APPENDIX G

Glossary

EBCDIC - Expanded Binary Coded Decimal Interchange Code. This code, used primarily in IBM equipment, is used to transmit information to peripheral equipment and other Computers. ASCII code is the Computer industry's standard and is similar. See "ASCII".

Edit - To change data, a program, or a program line.

Execution - The process which is performed by a Computer according to instructions.

Field - A set of related characters that make up a piece of data. For instance, a field of characters spelling a person's first name would be one field in a person's name and address record in a mail program's data file. See "Record" and "File."

File - A collection of related records that are treated as a unit. A file may contain data or represent a Computer program. A file can exist on diskette or hard disk. See "Field" and "Record."

Firmware - A Computer program that is part of the physical makeup of the Computer. See "Software" and "Hardware."

Fixed disk - See "Disk."

Flowchart - A symbolic representation of the logical flow of operations in a Computer program, usually very detailed.

Formatting - The process of organizing the surface of a diskette or disk to accept files of data and programs.

FORTRAN - FORmula TRANslator. A popular high-level programming language used primarily in scientific applications.

Graphics - This term generally refers to special characters which may be displayed or printed. In other uses, it indicates that the specified device may be able to reproduce any type of display, from photographs to line and bar charts. Often graphics' capabilities are expressed in pixels, or points which may be lit (number of points per row by number of rows).

APPENDIX G
Glossary

Hard copy - Typewritten or printed characters on paper, produced by a peripheral, called a printer.

Hard-sectored - This term applies to diskettes and indicates a type of diskette that has multiple timing holes which mark sector boundaries as well as the beginning of a track.

Hardware - The physical Computer and all of its component parts, as well as any peripherals and inter-connecting cables. See "Firmware" and "Software."

Hexadecimal - A numbering system based on sixteen and represented by the digits 0 through 9 and A through F. A single byte of data may be represented by two hexadecimal digits.

High level language - A programming language which uses symbol and command statements that an operator can read. Each statement represents a series of Computer instructions if expressed in machine language. Examples of high level languages are BASIC, COBOL, and FORTRAN.

Home - This term usually means the upper left-hand corner of the display screen, and specifically the first displayable character location.

I/O - Input/Output. This term refers to the devices which enter and/or store data and/or the paths through which such data passes. See "Port."

IC - Integrated Circuit- See "Chip."

Input - 1. Information or data transferred into the Computer. 2. The route through which such information passes. 3. The devices which supply a source of input data, such as the keyboard or disk drive.

Instruction - A program step that tells the Computer exactly what to do for a single operation in a program.

Integer - A whole entity (number). Not a part, fraction, or a number with a decimal point.

APPENDIX G

Glossary

Interactive processing - An operation where data is processed by a Computer under the supervision of an operator, often requiring many intermediate keyboard entries. See "Batch Processing."

Interface - A device that serves as a common boundary between two other devices, such as two Computer systems or a Computer and peripheral. See "RS-232 Interface."

Interference - This is usually termed RF Interference, for Radio Frequency Interference, and in recent years has come to the attention of the FCC (Federal Communications Commission). Interference is the presence of unwanted signals in an electrical circuit. In radio and television, it causes noise, static, and picture distortion and disruption. The FCC ruled that Computers must meet certain standards with regard to the amount of interference they cause in nearby radios and televisions.

Interpreter - A special program that interprets (usually) the code in a high level language for use by the Computer. It performs an interpretation each time an instruction is executed. Usually, this results in slower operation as compared to a compiled Computer language. However, the process of testing and debugging an interpreted Computer program is much easier and faster. BASIC is a high level language that is usually found in an interpreter form.

Interrupt - A temporary suspension of processing by the Computer and possible override by a high priority routine caused by input from another part of the Computer or a peripheral.

Jump - A departure from the normal sequential line-by-line flow of a program. A jump may be either conditional — based upon the outcome of a test — or unconditional (i.e., absolute).

Justify - To adjust exactly — the perfect alignment of a margin. Normal text applications are left justified — that is, the left margin is always aligned. A feature of many word processors is right justification where the right margin is also perfectly aligned by adding extra spaces between words or increments of a space between letters.

APPENDIX G
Glossary

K - The symbol used to equal 1024. Also the abbreviation of kilo, which stands for 1000. However, in Computers it is the power of two closest to the number (2 ^ 10); hence, the amount of 1024. As an example, 16K would equal 16 times 1024, or 16384. See "kilo."

Keyboard - A device used to enter information into a Computer. It is made up of two or more keys, often grouped as is a typewriter and/or calculator keyboard.

Keypad - A small keyboard or section of a keyboard containing a group of 10, 12, or 16 keys, generally those used on simple calculators.

Keyword - This is a single word in a high-level language that defines the primary type of operation to be performed.

Kilo - A prefix meaning one thousand. In Computers, it is abbreviated as K and also may refer to the power of two closest to a number — 4,096 is 4K. See "K."

Kilobyte - 1,024 bytes. See "Byte."

Language - A defined set of characters which, when used alone or in combinations, form a meaningful set of words and symbols. When we are speaking of a Computer language, we mean a set of words and operations, and the rules governing their usage. Examples of Computer languages are Machine Language, Assembler Language, BASIC, COBOL, and FORTRAN.

Load - The process of entering information (data or a program) into a Computer from keyboard, diskette, or other source.

M - Abbreviation for Mega. See "Mega."

Machine language - A programming language consisting only of numbers or symbols that the Computer can understand without translation.

APPENDIX G

Glossary

Main frame - 1. The actual central hardware of a Computer, containing the Central Processing Unit (CPU). 2. A large, multi-tasking, multi-user Computer, usually associated with financial and government institutions and having the ability to process very large amounts of data in a batch processing mode.

Maintenance - The process of maintaining hardware and software. With hardware, in addition to corrective maintenance or repair, this also includes preventive maintenance, or cleaning and adjustment. With software, maintenance refers to updating critical tables and routines to maintain accountability with established standards (as updating tax tables for Income and Social Security tax deductions in a payroll program).

Matrix - 1. A rectangular array of datum, usually numeric, subject to mathematical operations or manipulation. Any table is a matrix. 2. A rectangular array of elements which, when used in combination, may form symbols and/or characters, as in a dot-matrix printer or video display.

Mega - A term meaning one million. Abbreviated M. When used in Computers, it usually means one thousand K. One Megabyte equals 1,000 Kbytes, or 1,024,000 bytes.

Megabyte - 1,024,000 bytes. See "Mega."

Memory - A portion of a Computer that is used to store information (either data or programs). The size of a microcomputer is often determined by the amount of user memory (measured in Kilobytes) in the system. See "RAM," "ROM."

Microcomputer - A term that applies to a small, (usually) desktop Computer system, complete with hardware, software, and peripherals. See also "Minicomputer" and "Main Frame."

Minicomputer - A term that applies to medium sized Computer systems. See "Microcomputer" and "Main Frame."

Mnemonic - A term applying to an abbreviation or acronym that is easy to remember.

Mode - Method of operation. For instance, BASIC has two modes of operation: Direct Mode and Indirect Mode.

APPENDIX G
Glossary

Modem - MOdulator DEModulator. A device that converts the digital signals from a Computer into a form compatible with transmission facilities and vice-versa. Used most commonly with telephone communications.

Modulo - A mathematical operation resulting in the remainder of a division operation. 42 modulo 5 = 2 (the remainder of 42 divided by 5).

Monitor - 1. A control program in a Computer. 2. A black and white or color (CRT) display.

Multi-processing - A term which means doing two or more processes at the same time. While this usually applies to Computers with more than one CPU, it sometimes also applies to time-sharing. See "Time Share."

Multi-tasking - Often used synonymously with multi-processing, this term means doing two or more tasks at the same time. Further, as differing from multi-programming, which deals with unrelated tasks, multi-tasking is related and often deals with the same disk files.

Network - A network is the interconnection of a number of points by means of communications facilities, such as the telephone.

Numeric - Composed of numbers. The value of a number as contrasted to a character representation.

Octal - A numbering system based on eight and represented by the digits 0 through 7. A single byte of data may be represented by three octal digits.

OS - Operating System - A program or programs that provide basic utility operations and control of a Computer system.

Operation - A defined action; the action specified by a single Computer instruction.

Operator - The person who actually manipulates the Computer controls, places the diskette into the disk drive, removes printer output, etc.

Output - The results of Computer operations; this may be in the form of displayed or printed information, or data stored on, (for example) a diskette.

APPENDIX G

Glossary

Parallel - In Computers, this refers to information which is sent as a group, rather than serially. For instance, the eight bits of a byte are transmitted simultaneously over eight channels or wires. See "Serial."

Parameter - A specification or value used in an operation or statement.

Parity - Refers to a method used to check the validity of data that is stored, transmitted, or manipulated. The value of a Parity bit (which is added to the number of bits which make up one character) will be determined by the desired outcome of the sum of the bits for that character (i.e., to be either an odd or even number).

Peripheral - A device that is connected to the Computer for the purpose of supplying input and/or output capability to that Computer. A peripheral is also not under direct control of the Computer; it may be capable of some independent operation (self test, etc.).

Port - The path through which data is transferred into and/or out of the Computer.

Precision - The degree of exactness, often based upon the number of significant digits in a value.

Printer - A device used to produce Computer output in the form of (type)written or printed characters and symbols on paper. The output of a printer is called "hard copy" or a "Computer printout".

Problem - A situation where an unknown exists among a given set of knowns. The finding of the unknown might be assigned as the objective of a program or task.

Process - The act of completing or executing an instruction or set of instructions. It may include compute, assemble, compile, interpret, generate, etc.

Processor - A Computer or its CPU. See "CPU."

Program - A set of Computer instructions which, when followed, will result in the solution to a problem or the completion of a task.

APPENDIX G

Glossary

Program language - Any one of a number of languages created for a Computer. Examples include BASIC, COBOL, FORTRAN, and Assembly Language.

Programmer - A person who prepares and writes a Computer program.

Prompt - A symbol, character, or other sign that the Computer is waiting for some form of operator input. The prompt may request data and be made up of a query, requesting specific data. In other instances, the prompt may simply mean that the Computer is finished executing the latest command and is waiting for new instructions in the form of a command.

Pseudo - A prefix meaning an arbitrary substitution for.

Query - A specific request for data, usually accompanied by an operator prompt.

RAM - Random Access Memory. Volatile read-write memory in which data may be written to (stored) or read from (retrieved) directly. See "Random Access," and "volatile."

Random access - This term refers to the ability to access locations without regard to sequential position; that is, access may be accomplished by going directly to the location. On occasion, this is called "direct access."

Read - The process of obtaining data from some source, such as a diskette.

Read/write head - This is a magnetic recording/playback head similar to those used by tape recorders. The function of the head is to read (playback) and write (record) information on magnetic material such as disks or diskettes.

Real time clock - This portion of the Computer maintains a time function which may be used for making a record of the time used to complete an application. In many small Computers, this is a function of software, rather than hardware, and is subject to timing interrupts caused by certain operations.

Reset - The process of restoring the equipment to its initial state; which was reached by applying power to the system and turning it on.

APPENDIX G
Glossary

ROM - Read Only Memory. Memory which is similar to RAM, except that data cannot be written to it. Data can be read from it directly, as in the case of RAM, but ROM is non-volatile; that is, it will retain the information stored in it whether power is applied or not. It is most often used for special programs such as the monitor program in the Z-100 Desktop Computer. See "Volatile," "RAM," "PROM," "EPROM," and "EEPROM."

Routine - A sequence of instructions that carry out a well-defined function. A program may be called a routine, although programs usually contain many routines. If a routine is separated from the main body of the program it is referred to as a "subroutine."

RS-232 interface - A standardized interface adopted by the Electronic Industries Association (EIA) to ensure uniformity of interfacing signals between Computers and peripherals. This capability is built into most Computer devices. See "Interface."

Search - The systematic examination of data to locate a specific item. Searches are characterized by several different methods including sequential (items are examined in a specific sequence) and binary (ordered data containing the desired item is repeatedly halved until only the desired item remains).

Sector - A portion of a disk track. The location of a particular sector on the disk track is a matter of timing. In a diskette, timing is handled by timing holes. Diskettes containing only one timing hole are said to be soft-sectored because the timing is handled by software. Diskettes containing many timing holes are said to be hard-sectored because the timing is handled by hardware. See "Track."

Sequential - The order in which things follow, one after the other.

Serial - Refers (as referenced to data in computers) to data that has been broken down into a component part (character or bit) and handled in a sequential manner.

APPENDIX G
Glossary

Sign- An indication of whether the value is greater than zero (>0) or less than zero (<0). The dash or hyphen (-) is used to indicate a negative (less than zero) value. The absence of the dash or a plus sign (+) indicates a value greater than zero (positive).

Single density - This term is most often applied to the storage characteristics of diskettes, and generally refers to the density of the storage of bits on the diskette surface on each track. It also refers to the density of the diskette tracks, though this is not the common usage.

Software - This is a general term that applies to any program (set of instructions) that may be loaded into a Computer from any source. See "Firmware" and "Hardware."

Sort - To arrange (or place in order) data according to a pre-defined set of rules.

Syntax - The rules governing the use of a language.

System - An assembly of components into a whole — A Computer system is made up of the Computer plus one or more peripheral devices.

Table - A collection of data into a form suitable for easy reference. This glossary could be called a table.

Task - A job, usually to solve a problem or follow a specific set of instructions.

Telecommunications - This term refers to the transmission and/or reception of signals by wire, radio, light beam, telephone, or any other electronic means.

Terminal - An Input/Output device, usually consisting of keyboard and display screen. A terminal also may consist of a printer and keyboard; this is referred to as a "printing terminal." Either type may include a modem (either acoustic coupler type or the direct-connect type) for remote operation. Some (usually older models) may also include a paper tape punch and reader.

APPENDIX G

Glossary

Time share - The process of accomplishing two or more tasks at (apparently) the same time. The Computer will process one task at a time, but only a small portion before switching to the next. Because a Computer can process a great amount of data in a very short time, the switching between tasks is transparent to human observation except when many tasks are executed at the same time.

Track - The portion of a disk that one read/write head passes over while in a stationary position. Track density is measured in TPI (Tracks Per Inch).

Utility - A program that accomplishes a specific purpose, usually quite commonly needed by a wide range of applications. Most utilities are furnished free with a Computer system, while some, like sort routines, are sold by various vendors.

Variable - This term applies to an assigned memory location (represented by a symbol or name) where a value is stored by a program. The maintenance of the variable is handled by the program.

Verify - The act of comparing original data against stored data to assure correctness of the data.

Word processing - The ability to enter, manipulate, correct, delete, and format text; an application which is widely used in microcomputers. Word processors are used to write letters; and to prepare documents such as magazine articles, manuscripts, manuals, and books; to name only a few of their applications.

Write - The process of recording data on some object, such as display terminal, diskette, or paper.

BIBLOGRAPHY

Instant BASIC, Brown, Jerald R., Dilithium Press, 1978.

BASIC BASIC: An Introduction to Computer Programming in BASIC Language, Coan, James S., Hayden Book Co., second edition, 1978.

Programming in BASIC for Personal Computers, Heiserman, David L., Prentice Hall, 1981.

Microsoft BASIC, Knecht, Ken, Dilithium Press, 1979.

The BASIC Handbook: An Encyclopedia of the BASIC Computer Language, Lien, David A., Compusoft Publishing, second edition, 1981.

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX