MICROSOFT™

Z-BASIC
(Z-DOS™)

595-3074-02

Printed in the
United States of America

data
systems

HEATH

MICROSOFT™

Z-BASIC
(Z-DOS™)

593-0040-02
CONSISTS OF

MANUAL
595-2825-02
FLYSHEET
597-2747-02

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 of modification thereof.

This publication could contain technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of this publication.

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

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

or call:

(616) 982-3884 Application Software/SoftStuff Products
(616) 982-3860 Operating Systems/Languages/Utilities

Consultation is available from 8:00 AM to 7:30 PM (Eastern Time Zone) on regular business days.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in paragraph (b) (3) (B) of the Rights in Technical Data and Computer Software clause in DAR 7-104.9(a). Contractor/Manufacturer is Zenith Data Systems Corporation, of Hilltop Road, St. Joseph, Michigan 49085.

Trademarks and Copyrights

Microsoft is a registered trademark of Microsoft Corporation.
The Microsoft logo is a trademark of Microsoft Corporation.
Z-DOS is a trademark of Zenith Data Systems Corporation.

Copyright© by Microsoft, 1982, all rights reserved.
Copyright© 1982 by Zenith Data Systems Corporation.

Essential Requirements for using Z-BASIC:

a. Distribution Media: One 5.25-inch soft-sectored 48-tpi disk
b. Machine Configuration (minimum): Z-100, 128K memory, one floppy disk drive, and CRT
c. Operating System: Z-DOS
d. Microcomputer Languages: Not applicable.

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

ZENITH DATA SYSTEMS CORPORATION
ST. JOSEPH, MICHIGAN 49085

IMPORTANT

NOTE: Z-BASIC supports a number of commands (such as LOCATE, CLS, SCREEN, and COLOR) that allow you to easily control features of the video display. Some previous versions of BASIC have not supported these commands, but instead used special terminal escape sequences.

When writing programs that require screen functions, use only the applicable, documented Z-BASIC commands. **Do not use** either terminal escape sequences or Z-BASIC editing commands (expressed as CHR$(n) functions) because they can produce unpredictable and undesirable results and they may not be supported in future releases of Z-BASIC.

CONTENTS

Part I: Introduction

Preface xii
Major Features of Z-Basic xiii

Chapter 1 **General Information**

Manual Organization 1.1
Physical Organization 1.1
Content Organization 1.2
Using the Z-BASIC Manual 1.3
General Overview of Languages 1.4
Interpreters 1.6
Compilers 1.7
The Program Development Process 1.8
Applications Programs 1.10

CONTENTS

Part II: BASIC Overview

Chapter 2 **Program Entry**

Starting Z-BASIC with the Z-DOS Operating System 2.1
Modes of Operation 2.4
  Direct Mode 2.5
  Indirect Mode 2.6
Character Set and Reserved Words 2.7
  Character Set 2.7
  Reserved Words 2.9
Line Format 2.10
Files and File Naming 2.12
Control Characters 2.17
Syntax Notation 2.18
Delimiters Used in Z-BASIC Printing 2.19
  The Comma 2.19
  The Semicolon 2.20

Chapter 3 **Editing Z-BASIC Programs**

The Full Screen Editor 3.1
  The Edit Command 3.2
  Inputting Z-BASIC Programs 3.2
  Changing a Z-BASIC Program 3.4
  Syntax Errors 3.4
  Logical Line Definition and INPUT Statements 3.5
The Full Screen Editor— Key Assignments 3.6

CONTENTS

Chapter 4 Programming in Z-BASIC

Loading the BASIC Interpreter 4.1
Writing a BASIC Program 4.4
Running a BASIC Program 4.6
Debugging a BASIC Program 4.7
Saving a BASIC Program 4.9
Loading a BASIC Program 4.10
Listing a BASIC Program to a Line Printer 4.11

Chapter 5 Arithmetic and String Operators

Variables 5.1
- Variable Names for Numbers and for Character Strings . 5.1
- Exceptions to Naming Variables 5.2
- Common Uses for Variables 5.3
- Declaring Variable Types 5.7

Array Variables 5.12
- Array Declarator 5.12
- Array Subscript 5.13
- OPTION BASE Statement 5.13
- Vertical Arrays 5.14
- Multi-Dimensional Arrays 5.14
- Matrix Manipulation 5.16
- Scalar Multiplication 5.17
- Transposition of a Matrix 5.17

Arithmetic Operators and Expressions 5.19
- Arithmetic Operators 5.19
- Relational Operators 5.27
- Logical Operators 5.32
- Numeric Functional Operators 5.46

Numeric Constants and Precisions 5.48
Converting Numeric Precisions 5.51
String Expressions and Operators 5.54

CONTENTS

Chapter 6 **File Handling**
File Manipulation and Management 6.1
File Manipulation Commands 6.1
Protected Files 6.3
File Management Statements 6.3
Sequential Data Files 6.5
Creating a Sequential Data File 6.7
Adding Data to a Sequential Data File 6.14
Random Access Files 6.16
Creating a Random File 6.18
Opening a File for Random Access 6.19
Structuring the Random Buffer into Fields 6.20
Assigning Data to Fields and Writing the Buffer to the Disk 6.21
Getting Records Out of the File 6.24
Storage and Retrieval of Numeric Data 6.26

Chapter 7 **Plotting Coordinates**
The Video Screen 7.1
Screen Statements 7.3
SCREEN Function 7.5
Locating and Activating Pixels 7.7
PSET Statement 7.9
PRESET Statement 7.11
Changing the Cursor Position 7.12
CSRLIN and POS Function 7.14

Chapter 8 **Advanced Color Graphics**
Using Color Graphics 8.1
The Video Board 8.1
The COLOR Statement 8.2
LINE, CIRCLE and PAINT Statements 8.5
The LINE Statement 8.5
The CIRCLE Statement 8.9
The PAINT Statement 8.12
GET, PUT, and DRAW Statements 8.14
The DRAW Statement 8.14
Movement Commands 8.15
GET and PUT Statements 8.22
Z-BASIC Summary Program 8.30

CONTENTS

Chapter 9 **Basic Language Summary**

Commands 9.1
Statements 9.3
    Data Type Definition Statements 9.3
    Assignment and Allocation Statements 9.3
    Control Statements 9.4
    Conditional Execution Statements 9.5
    NON-I/O Statements 9.5
    I/O Statements 9.6
Functions 9.8
    Arithmetic Functions 9.8
    String Functions 9.9
    Special Functions 9.10
    Variables 9.11
    Color and Graphic Statements 9.12

Part III: Reference Guide

Chapter 10 **Alphabetical Reference Guide**

Part IV: Appendices, Glossary and Index

APPENDIX A: Error Messages A.1
APPENDIX B: Converting Programs to Z-BASIC B.1
APPENDIX C: ASCII Character Codes and H-19 Graphic Symbols C.1
APPENDIX D: Mathematical Functions D.1
APPENDIX E: Assembly Language Subroutines E.1
APPENDIX F: Communications I/O F.1
Glossary G.1
Bibliography H.1
Index X.1

CONTENTS

List of Tables

1.1: Comparison BASIC vs. Assembly 1.5
2.1: Imput and Output Devices 2.13
3.1: Full Screen Editor Key Values 3.7
5.1 Precision Declaration on Various Values 5.11
5.2: Array Storage Allocation 5.14
5.3: Multi-Dimensional Array Storage Allocation 5.15
5.4: Order of Precedence 5.20
5.5: Algebraic Expressions and Their BASIC Counterparts . 5.23
5.6: Relational Operators 5.27
5.7: Negative Meaning of Relational Operators 5.28
5.8: Negated Structure of Relational Operators 5.28
5.9: Truth Table 5.33
5.10: DeMorgan's Laws 5.34
5.11: IMP Operator 5.36
5.12: Bit Pattern Equivalence 5.40
5.13: Numeric Functions 5.47
6.1: File Management Statements 6.3
6.2: Sequential File Statements and Functions 6.7
6.3: Creating a Sequential File — Program Steps 6.8
6.4: Random File Statements and Functions 6.17
6.5: Program Steps for Creating a Random File 6.18

CONTENTS

List of Figures

1.1: Program Development Process 1.9
3.1 Function Keys 3.10
3.2 Alphanumeric Keys 3.11
3.3 Keypad 3.11
3.4 Special Keys 3.11
7.1 X,Y Coordinates of the Four-Corner Points 7.2
8.1 Angles of a Circle 8.9

PREFACE

BASIC is a high-level computer programming language specifically designed for use by people with little programming experience, as well as experienced computer programmers. The name stands for *Beginner's All-purpose Symbolic Instruction Code*. As you begin to write and modify programs or develop your own software, you will appreciate BASIC's features.

BASIC's program commands and statements use ordinary English words such as PRINT, LIST, and EDIT. Its numeric calculations resemble elementary algebraic operations. These familiar terms make BASIC easy to learn, remember, and use.

As a high-level language, BASIC accomplishes many functions with just a few program statement lines. BASIC is an *interactive* language, which permits you to enter data and modify programs while they are being developed.

The BASIC interpreter translates your program into machine code that the computer understands. The interpreter's job includes analyzing your programs, checking for errors, and performing the functions you request. The BASIC interpreter assists in debugging programs and often pinpoints errors before the code is stored. The usability factor of BASIC has made it a very popular microcomputer language.

There are many technical advantages of BASIC. It supports most printers and disk peripherals. Although various versions and "dialects" of BASIC exist, one version can usually be adapted to another. Additionally, BASIC programs are virtually machine independent and will run on most computer systems with few modifications.

The version of BASIC referenced in this manual is called Z-BASIC. Z-BASIC has many more commands and features than previous versions of BASIC. These new commands will assist you in your efforts to create useful BASIC programs.

PREFACE

Major Features of Z-BASIC

Z-BASIC has many features that will assist you in creating useful programs. Here are some of the enhanced features provided in this version.

1. Four variable types: Integer (+32767), String (up to 255 characters), Single-Precision Floating Point (7 digits), Double-Precision Floating Point (16 digits).

2. Trace facilities (TRON/TROFF) for easier debugging.

3. Error trapping using the ON ERROR GOTO statement.

4. PEEK and POKE functions to read from and write to any memory location.

5. Automatic line number generation and renumbering, including automatic changing of referenced line numbers.

6. Arrays with up to eight dimensions.

7. Boolean operators OR, AND, NOT, XOR, EQV, and IMP.

8. Formatted output using the complete PRINT USING facility, including asterisk fill, floating dollar sign, scientific notation, trailing sign, and comma insertion.

9. Direct access to the 256 I/O ports with the INP and OUT functions.

PREFACE

10. The Full Screen Editor and the extensive program editing facilities via EDIT command and EDIT mode subcommands.

11. Assembly language subroutine calls (up to 10 per program) are supported.

12. IF/THEN/ELSE and nested IF/THEN/ELSE constructs, and WHILE/WEND and nested WHILE/WEND constructs.

13. Variable length random and sequential disk files with a complete set of file manipulation statements: OPEN, CLOSE, GET, PUT, KILL, and NAME.

14. Event trapping which allows a program to trap the occurrence of a specific communication event by trapping a specific line number.

15. Advanced graphic techniques including; LINE, CIRCLE, GET, PUT, and DRAW statements.

16. RS-232 support.

17. Time and Date setting and retrieval.

CHAPTER 1 GENERAL INFORMATION

Manual Organization

BRIEF

The content of this manual is organized into four convenient parts:

PART 1.	Introduction
PART 2.	BASIC Overview
PART 3.	BASIC Reference Guide
PART 4.	Appendices, Glossary and Index

The information throughout this manual is physically structured according to the following format:

- Brief
- Details
- Checkpoint
- Application

Details

PHYSICAL ORGANIZATION

Brief

Brief is a short description of the key points covered in the section. It is located at the beginning of each section. Those of you who are experienced users can use this section as a concise reminder of the options available, and then continue reading only if you find it necessary for a clearer understanding or for specific information. The brief is also valuable to the beginner to use as a preview of the upcoming information.

Details

Details is an easy-to-follow explanation of all the information covered in the section. Step-by-step procedures, sample programs, comparisons, and analogies may be present in this section. This information was specifically developed with the new user in mind; but if you are experienced, this portion of the text may clarify a concept or refresh your memory.

GENERAL INFORMATION

Manual Organization

Checkpoint is the vehicle used to test your comprehension of the material presented. It may contain information on how to recover from an error that may have occurred while you were implementing previous instructions or it may contain a sample program you can input to illustrate how associated commands are integrated. It can also be a summary of the preceding material. Checkpoint is designed to summarize and "tie-up-the-loose-ends" and to test your understanding.

Checkpoint

Application is used when necessary to provide additional technical considerations or to provide a practical application of the material. Generally, this section will be a complex extension of what has been covered. This portion is included with experienced users in mind. However, if you are a beginner, you may find it useful if you are comfortable with your understanding of the material.

Application

CONTENT ORGANIZATION

Chapter 1, Page 1.4, gives an informational overview of languages. This chapter discusses high level languages, interpreters, compilers, and the program development process.

Chapter 2, Page 2.1, provides general information on entering BASIC programs. In this chapter, you will find information on how to start BASIC, the modes of operation, the character set and reserved words, how a program line is formed, control characters, delimiters, and notation used in BASIC.

Chapter 3, Page 3.1, provides all the information necessary to use the BASIC full screen editor.

Chapter 4, Page 4.1, is an overview of programming in BASIC. This section does not attempt to teach BASIC programming, but it does provide general information for getting started.

Chapter 5, Page 5.1, is a thorough discussion of arithmetic and string operators. In this chapter you will find information on variables, array variables, arithmetic operators and expressions, numeric constants and precisions, converting numeric precisions, and finally, string expressions and operators.

Chapter 6, Page 6.1, "File Handling", discusses file management, sequential-access, and random access disk operations.

Chapter 7, Page 7.1, discusses the graphic capabilities of Z-BASIC, the video screen, and the plotting of coordinates.

Chapter 8, Page 8.1, provides detailed information on the advanced color commands of Z-BASIC and how to use the color video display input and output commands.

Chapter 9, Page 9.1, lists each command, function, statement, and variable according to its function within a program. Following that, in Chapter 10, is the *Alphabetical Reference Guide*, where each Z-BASIC statement, command, function and variable is referenced in alphabetical order.

Following the Alphabetical Reference Guide are the Appendices, Glossary and an Index. The Appendices provide specific information on the topics that follow:

Error Messages
Converting Programs to Z-BASIC
ASCII Character Codes and H-19 Graphic Symbols
Mathematical Functions
BASIC Assembly Language Subroutines
Communication I/O

For specific information pertaining to the operation and capabilities of the Z-100 Desktop computer, refer to the Z-100 User's Manual. Also within the Z-100 User's Manual is information relative to the care and handling of disks.

Additionally, there are programming concepts in the User's Manual with which you may want to familiarize yourself before reading this manual.

For infomation pertaining to the operational characteristics of the Z-DOS™ operating system, refer to the Z-DOS documentation.

USING THE Z-BASIC MANUAL

Some features of Z-BASIC are new even to the most experienced user. To help facilitate easy understanding, we have included many program examples. We suggest that you read the chapters, study the examples and then input them on your computer to watch the visual effects. Then, if you feel comfortable with your understanding, try modifying the programs. This way you will be able to see and take full advantage of Z-BASIC's capabilities.

™Z-DOS is a trademark of Zenith Data Systems Corporation.

GENERAL INFORMATION

General Overview of Languages

BRIEF

Programming languages provide a means of communication between computers and users.

The computer interprets symbols (binary code) in order to know what instructions to execute.

Languages interpret and/or compile English terms and mnemonic codes into binary codes so the computer can understand and execute the instructions.

The program development process involves creating a BASIC source file, debugging and executing the programs.

Details

The following section explains the need for languages, how they are used in general, and what interpreters and compilers actually do.

Why Languages Exist

Computer languages are available to provide clear, direct and efficient communication between people and computers. As with human languages, computer languages have their own dialect, grammar, and syntax. There are hundreds of different computer languages and language dialects that can be classified into three (sometimes overlapping) categories. They are: machine language, assembly languages, and high-level languages. Z-BASIC is a high-level language.

The development of languages was spurred by programmers who wanted to use previously written and debugged programs developed by others. This was often very difficult because of differences in notation, levels of precision, and differences in the way parts of programs were linked together. It became necessary to develop a library of facilities and routines, as well as the capability of easily linking parts of programs together.

Additionally, there was a demand for the capability of writing programs in a computer shorthand. Programmers wanted shorter and more natural notation, which was not available in machine language. Programmers wanted a language which was more natural, and like English, that would make expressing ideas simpler and more concise.

GENERAL INFORMATION

General Overview of Languages

In response to these demands, high-level languages were developed. A *programming language* can be defined as the rules for combining a set of symbols or symbolic expressions into meaningful communication between a person and a computer.

Advantages of Using High-Level Languages

One advantage of using a high-level language is that you do not have to know machine code to write a program. It is sometimes helpful to know about such things as memory allocation, addresses, input and output ports, and how numbers are represented internally, because this knowledge can help you develop your programs more efficiently. However, it is not necessary to understand all of these hardware concepts before you begin learning a high-level language.

Another advantage is that programs written in high-level languages are, for the most part, *machine independent*. They have the potential to be transferred to another computer using the same language with little modification of the code.

Most programmming languages have a problem-oriented notation that makes them easier to learn than machine code. *Problem oriented* means the statement of the problem in code is relatively close to the statement of the problem in English or arithmetic terms. For example, IF A=B+C THEN 100 is much easier to understand than the equivalent equation written in assembly language (see the comparison in Table 1.1). This factor makes coding and the understanding of written codes easier.

BASIC Statement	vs	Assembly Statement
IF A=B+C THEN 100		PUSH PSW
		MOV A,B
		ADD C
		MOV B,A
		POP PSW
		CMP B
		JZ L100

Table 1.1
A comparison between a BASIC Statement and an equivalent assembly language statement

GENERAL INFORMATION

General Overview of Languages

Interpreters and Compilers

A microprocessor can execute only its own machine instructions; it cannot execute BASIC statements directly. Therefore, before a program can be executed, some type of translation must occur from the statements contained in your BASIC program to the machine language of your microprocessor. Compilers and interpreters are two types of programs that perform this translation. This manual is the documentation for the Z-BASIC interpreter; however, the following discussion explains the difference between these two translation schemes, and explains why and when you would want to use the compiler.

INTERPRETERS

Generally, an *interpreter* translates your BASIC program line by line during program execution. To execute a BASIC statement, the interpreter must analyze the statement, check for errors, and then perform the BASIC function requested.

If a statement is executed repeatedly, this interpretive process is repeated each time the statement is executed.

During interpretation, BASIC programs are stored as a list of numbered lines. Each line is not available as an absolute memory address. Therefore, branch commands such as GOTO and GOSUB cause the interpreter to search for line numbers.

Additionally, the interpreter maintains a list of all variables. When a BASIC statement refers to a variable, the interpreter searches this list of variables to find the referenced variable. (Absolute memory addresses are not usually associated with the variables in interpreted programs.)

COMPILERS

A *compiler* translates a source program and creates a new file called an object file. The object file contains code that can be read by the computer. All translation takes place before run time; no translation of your BASIC source file occurs during the execution of your program. In addition, absolute memory addresses are associated with variables and with the targets of GOTO and GOSUB commands, so that lists of variables or of line numbers do not have to be searched during execution of your program.

Note also that a compiler can be an optimizing compiler. *Optimization* is a process by which a program is continually adjusted to achieve the best obtainable set of operating conditions. Optimizations such as expression re-ordering and sub-expression elimination are made to either increase the speed of execution or to decrease the size of your program.

It is important to remember that you do not need a compiler to develop or execute BASIC programs. It is defined here so you will know the function it serves when it is used and its relationship to the interpreter.

GENERAL INFORMATION

THE PROGRAM DEVELOPMENT PROCESS

This discussion of the program development process is keyed to Figure 1.1 which is a flowchart that illustrates the process. You may find it useful to refer to the Figure when reading this text.

1. Program development begins with the creation of a BASIC source file. The best way to create a BASIC source file is with the editing facilities of BASIC, although you can use any general purpose text editor.

2. Once you have written a program, you can use BASIC to debug the program by running it to check for syntax and program logic errors. In many instances the BASIC interpreter will "flag" errors for you with an error message indicating the line number the error is in and what type of error is present. However there are instances when the only indication of an error is an unexpected or undesired result. Correct the errors in your program and then run the program again.

3. If your program is totally debugged you may wish to compile it. If you intend to use a BASIC compiler, some additional steps are required. Refer to a BASIC compiler manual for this information.

The flow chart shown below illustrates the development process of a BASIC program from the creation of the program with the BASIC interpreter through the process of compilation.

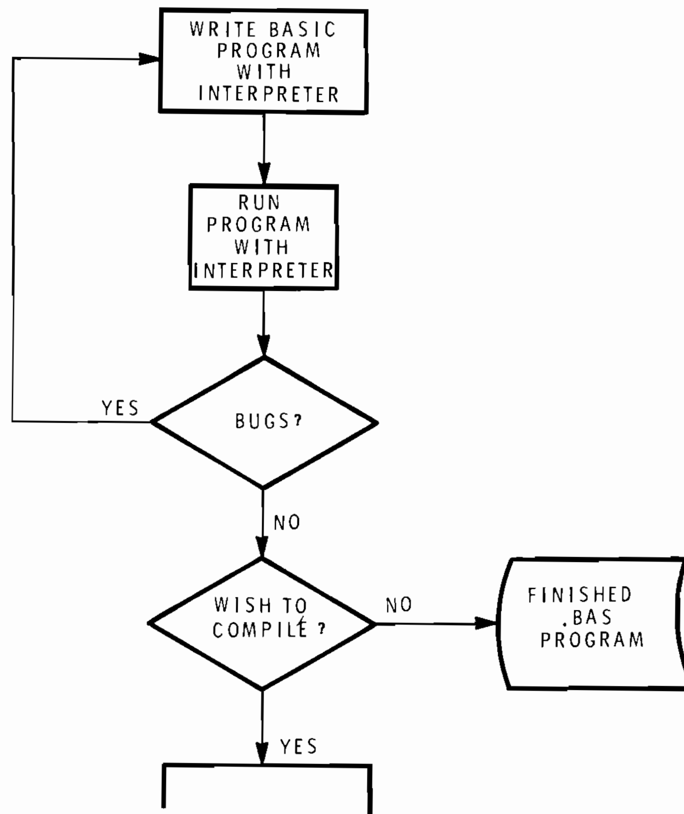


Figure 1.1
The Program Development Process

APPLICATION PROGRAMS

The final topic in the overview of languages is application programs. An application program performs functions for the user directly. Unlike the previously discussed programs that compile, interpret, or in some way support other programs, application programs actually perform the work you want done.

Examples of what application programs can do include: prepare payrolls, manage inventory, maintain and update records, and track purchasing. The application program reads and processes data from the system software and puts it into an easily understood and accessible format.

Starting Z-BASIC with the Z-DOS Operating System

BRIEF

Z-BASIC is loaded into memory by typing the command:

A: **ZBASIC**

The format of the Z-BASIC command line with options is:

```
ZBASIC [<filename>]
[/M:<highest memory location>]
```

Details

Z-BASIC is loaded and executed by typing ZBASIC in response to the Z-DOS command line prompt: A:.

After loading, Z-BASIC responds with the following:

```
Z-BASIC rev. 1.0
[Z-DOS/MSDOS version]
Copyright 1982 (C) by Microsoft
Created: 20-AUG-82
xxxxx Bytes free
Ok
```

The Ok means that Z-BASIC is ready to accept your commands.

The Z-BASIC operating environment may be altered by specifying options following Z-BASIC on the command line. It is important to remember that it is not necessary to specify these options to start using Z-BASIC. The format of the Z-BASIC command line with options is:

```
ZBASIC [<filename>]
[/M:<highest memory location>]
```

PROGRAM ENTRY

Starting Z-BASIC with the Z-DOS Operating System

Filename

If <filename> (the file name of a BASIC program) is present, BASIC proceeds as if a RUN <filename> command were given after initialization is complete. A default file extention of .BAS is assumed if none is given. This allows BASIC programs to automatically run by putting this form of the command line in a Z-DOS AUTOEXEC.BAT file. Programs run in this manner will need to exit via the system in order to allow the next command from the AUTOEXEC.BAT file to be executed.

Number of Files

There is no longer a need to specify the number of files, maximum record size or maximum buffer size which were optional specifications in previous versions of BASIC.

Maximum Record Size

In Z-BASIC, disk buffers are allocated dynamically, meaning, the length of the allocated workspace is automatically determined by how much space your program requires. Record lengths may range from 1 to 65535 bytes. The maximum number of files that can be open at one time is 255 files.

Buffer size

The COM1: device buffer is fixed at 120 bytes.

Starting Z-BASIC with the Z-DOS Operating System

Highest Memory Location

The /M:<highest memory location> switch sets the highest memory location that will be used by BASIC. BASIC will attempt to allocate 64K of memory for the data and stack segments. If machine language subroutines are to be used with BASIC Programs, use the /M: switch to reserve enough memory for them.

NOTE: <highest memory location> may be specified in decimal, octal (preceded by &O), or hexadecimal (preceded by &H).

Examples:

`A: ZBASIC PAYROLL`	Use all of memory, load and execute PAYROLL.BAS.
`A: ZBASIC /M:32768`	Use the first 32K of memory.

PROGRAM ENTRY

Modes of Operation

BRIEF

When BASIC is at the command level, you can use it in either the direct mode or the indirect mode.

In the direct mode, statements and commands are executed immediately and the results can be stored for later use. However, the instructions are not saved.

When BASIC is in the direct mode, statements and commands are not preceded by line numbers.

The direct mode is especially useful for routine mathematical calculations that do not require a program or to test the use of commands that are unfamiliar to you.

The indirect mode is used for running BASIC programs.

In the indirect mode, program lines are preceded by line numbers and are stored in memory.

Details

When BASIC is initialized, it displays the sign-on information discussed in the previous section and then types the prompt `Ok`. `Ok` indicates BASIC is at the command level and is ready to accept commands. At this point, you can use BASIC in either the direct mode or the indirect mode.

DIRECT MODE

The *direct mode* is useful for learning BASIC, debugging programs and for using BASIC as a calculator for quick computations that do not require a complete program.

Here are some examples of expressions written in the direct mode:

```
PRINT 4*8
PRINT -3/6
PRINT -2+5
PRINT (2-3)+(6.5*9.2)
```

In the direct mode, also known as the *immediate mode*, BASIC statements and commands are not preceded by line numbers. They are executed when they are entered (when the RETURN key is pressed).

In the direct mode, results of arithmetic and logical operations may be displayed immediately or stored for later use, but the instructions themselves are lost after execution.

Variables are changeable quantities that are represented by a symbol or name. These variables can be assigned to specific values in the direct mode as follows:

```
LET A = 1
LET B = 2
LET A = A+B

PRINT A

LET C = B
PRINT (A*B)+(B*C)
LET C = C+1
PRINT (A*C)+(B*C)
```

LET is an optional statement (also covered on Page 10.88) that allows you to assign specific values to variables. These assignments stay in effect as long as you are in the direct mode and can be subsequently used in other expressions as shown in the example above.

PROGRAM ENTRY

Modes of Operation

In the first PRINT statement on the previous page, "PRINT A", the value for A is 3. Notice that the statement LET A=A+B means "add the present values of A and B and assign the sum to A."

The second PRINT statement "PRINT (A*B)+(B*C)" equates to 10 because (3*2)+(2*2)=6+4=10.

The third PRINT statement in the example above equates to 15 because (3*3)+(2*3)=9+6=15. Notice that the statement LET C=C+1 means, in effect, "increase the stored value of C by 1."

NOTE: Using RUN will set all variables to zero, including those that have been previously set in the direct mode, and any variables from a previous run. If you want to execute your program without clearing the variables, use the GOTO statement in the direct mode.

INDIRECT MODE

The *indirect mode* is normally used for entering programs that will be run more than once. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed when you enter the RUN command. Here is an example of a BASIC program written in the indirect mode.

```
10 LET A=2
20 LET B=3
30 PRINT A+B
RUN
 5
Ok
```

You can think of the preceding example as a sequential program that is a series of immediate mode statements in which each line has been prefaced by a line number. Such statements are said to be in indirect mode because the computer defers execution until a RUN command is entered, instead of executing the program lines immediately.

PROGRAM ENTRY

Character Set and Reserved Words

CHARACTER SET

BRIEF

The BASIC character set is comprised of alphabetic characters, numeric characters, and special characters.

The alphabetic characters in BASIC can be either capital or lower-case letters.

The numeric characters in BASIC are the digits zero through nine.

Reserved words are words that have a special meaning in BASIC including BASIC commands, statements, functions, and operators.

They may not be used as variable names.

In order for reserved words to be recognized by BASIC, they must be delimited by spaces or special characters as allowed by syntax.

Details

The following special characters are recognized by BASIC:

Character	Name
	Blank space
;	Semicolon
=	Equal sign or assignment symbol
+	Plus symbol
–	Minus symbol or dash
*	Asterisk or multiplication sign
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number or pound sign

PROGRAM ENTRY

Character Set and Reserved Words

Character	Name
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period
'	Single quotation mark (apostrophe)
"	Double quotes
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At-sign
_	Underscore
{	Left brace
}	Right brace
DELETE	Deletes last character typed.
ESC	Escapes edit mode subcommands.
TAB	Moves print position to the next tab stop. Tab stops are every eight columns.
LINE FEED	Moves to the next physical line.
RETURN	Terminates input of a line.

RESERVED WORDS

All of the reserved words recognized by BASIC are listed below:

ABS	DATE$	HEX$	MKD$	RENUM	USR
AND	DEF	IF	MKI$	RESET	VAL
ASC	DEFDBL	IMP	MKS$	RESTORE	VARPTR
ATN	DEFINT	INKEY$	MOD	RESUME	WAIT
AUTO	DEFSNG	INPUT	NAME	RETURN	WEND
BEEP	DEFSTR	INPUT$	NEW	RIGHT$	WHILE
BLOAD	DELETE	INPUT#	NEXT	RND	WIDTH
BSAVE	DIM	INP	NOT	RSET	WRITE
CALL	DRAW	INSTR	NULL	RUN	WRITE#
CDBL	EDIT	INT	OCT$	SAVE	XOR
CHAIN	ELSE	KEY	ON	SCREEN	
CHR$	END	KILL	OPEN	SGN	
CINT	EOF	LEFT$	OPTION	SIN	
CIRCLE	EQV	LEN	OR	SPACE$	
CLEAR	ERASE	LET	OUT	SQR	
CLOSE	ERL	LINE	PAINT	STEP	
CLS	ERR	LIST	PEEK	STOP	
COLOR	ERROR	LLIST	POINT	STR$	
COM	EXP	LOAD	POKE	STRING$	
COMMON	FIELD	LOC	POS	SWAP	
CONT	FILES	LOCATE	PRESET	SYSTEM	
COS	FIX	LOF	PRINT	TAN	
CSNG	FNxxxxxxxx	LOG	PRINT#	THEN	
CSRLIN	FOR	LPOS	PSET	TIME	
CVD	FRE	LPRINT	PUT	TIME$	
CVI	GET	LSET	RANDOMIZE	TO	
CVS	GOSUB	MERGE	READ	TROFF	
DATA	GOTO	MID$	REM	TRON	
DATE				USING	

PROGRAM ENTRY

Line Format

BRIEF

The line format of program lines in BASIC is:

nnnnn BASIC statement[:BASIC statement...]['comment]RETURN

nnnnn indicates the line number which can be from one to five digits.

You may have more than one statement on a line, but each statement must be separated by a colon.

You can add comments to the end of the line by using the ' (single quote) or :REM to separate the comment from the rest of the line. The single quote does not require a preceding colon.

Details

Line Numbers

To enter a program line into a BASIC program in the indirect mode, you must first type a line number, which can be from one to five digits. Line numbers are used to show the order in which the program lines are stored in memory and also are used as reference points for branching and editing. Line numbers must be in the range from 0 to 65529. The line number is followed by a BASIC statement, which can be a command, statement, function, or variable. A *statement* is a meaningful expression or an instruction in a source language. Each statement is followed by a RETURN.

Example:

```
Ok
10 FOR I=1 TO 10
20 PRINT I
30 NEXT I
RUN
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
Ok
```

Statements

In the preceding example, 10, 20, and 30 are line numbers. Each line number is followed by a BASIC statement that contains instructions for the program. In this case line 20 is an instruction to print I, which is defined in line 10 as the numbers between 1 and 10. In line 30, NEXT is part of the format necessary when any FOR statement is used. See the Alphabetical Reference Guide, Pages 10.53 — 10.56, for additional information on FOR ... NEXT statements. RUN is the command used to execute a program.

Multiple Statements on a Line

You can have more than one BASIC statement on a line, but each statement must be separated from the last statement by a colon, except for the single quote for a remark at the end of the line. The total number of characters in the line must not exceed 255.

```
OK
10 FOR I=1 TO 10: PRINT I: NEXT I
RUN
```

Executable and Non-executable Statements

A BASIC statement is either executable or non-executable. *Executable statements* are program instructions that tell BASIC what to do during the execution of a program. In the above example, PRINT I is an executable statement. *Non-executable statements* do not cause any program action.

A remark statement or comment is an example of a non-executable statement. A comment, which is indicated by a single quote (') or the keyword REM, preceded by a colon, can be added to the end of any line. Comments are used to help make the program readable by explaining what is going on in that line. For example:

```
10 PRINT  "ENTER  YOUR  NAME";  'Ask user's Name
20 INPUT NAM$               'Get response from keyboard
30 PRINT "OK "NAM$ 'Print response on display screen
RUN
ENTER YOUR NAME?             JOHN DOE
OK JOHN DOE
Ok
```

Checkpoint

To test your understanding of line format, input the example on Page 2-10. Note that the line numbers are followed by BASIC statements. After you input lines 10, 20, and 30, type **RUN**. If the numbers 1-10 appear on your screen, you have input the sample correctly. If you receive a syntax error, check the sample again.

PROGRAM ENTRY

Files and File Naming

BRIEF

A physical file is identified by its file specification, or filespec for short. The filespec is a string expression which uses the following format:

device:filename.extension (e.g. A: Inventry.BAS)

The device name tells BASIC where to look for the file (which device — e.g. disk drive). The device name consists of one to four characters, followed by a colon (:).

The filename tells BASIC which file you are looking for, and may be up to eight characters long.

The extension usually identifies the file type and may be up to three characters long.

Details

Files

A *file* is a collection of related information treated as a unit. Information is stored in a file on a disk. In order to use the information, you must tell BASIC where the information is and then open the file. Then you may use the file for input and/or output.

Filespec

The file is described by its *file specification*, or filespec, which is a string expression with the following format:

device:filename.extension

Device Name

The device name tells BASIC where to look for the file. A device can be internal, such as an inboard disk drive in the Z-100, or it may be a peripheral device (a device that is connected to the computer and controlled by the computer). It is through these devices that input to and output from your file is possible. The specification of the device is optional. If the file you wish to open is in the default (current) drive, it is not necessary to specify the device name.

The device name consists of one to four characters followed by a colon (:). The following device name chart tells what device you use for input and output.

Device		I/O
KYBD:	Keyboard.	Input only
SCRN:	Screen.	Output only
LPT1:	PRN	Output only
Communication Devices		
COM1:	AUX	Input and Output
Storage Devices		
A:	Disk Drive#1	Input and Output
B:	Disk Drive#2	Input and Output
C:	Disk Drive#3	Input and Output
D:	Disk Drive#4	Input and Output
E:	Disk Drive#5	Input and Output
F:	Disk Drive#6	Input and Output

Table 2.1

Input and Output Devices

PROGRAM ENTRY

Files and File Naming

FILENAME

The filename tells BASIC which file you are looking for. The filename may consist of two parts, separated by a period (.) in the following format:

name.extension

The name is a character string that is from one to eight characters long. The extension, which usually indicates the type of file, may be no more than three characters long. If the extension is longer than three characters, the extra characters are truncated. *Truncation* means dropping the extra letters so that the filename will be in accordance with file naming conventions.

If you input a name that is longer than eight characters and the extension is not included, BASIC inserts a period after the eighth character and uses the extra characters (up to the third character) for the extension.

Recognized Characters

The characters that are recognized and acceptable to BASIC in name and extension are:

A through Z
0 through 9
$
@

Examples of filenames allowed in BASIC are:

01JAN82.YR
JDL
PROGRAM3.416
JOY.BAS
@$$@$$.213

The following examples illustrate how BASIC truncates names and extensions in accordance with file naming conventions when the names are too long.

B23335RS3JUTEW will be B23335RS.3JU
DISKETTE.BACKUP will be DISKETTE.BAC
@@WRONGWAY.BAS will cause an error message to be displayed

Checkpoint

In summary, a file is identified by its file specification, which is the device followed by a filename. A device name can be one to four characters followed by a colon and can be an input device, output device or both. A filename must conform to Z-DOS filename conventions; namely, the name must be from one to eight characters long and the extension can be no longer than three characters. When a filename is entered that is too long, BASIC will truncate that name if possible.

A default extension of .BAS is used on LOAD, SAVE, MERGE and RUN <filename> commands if no "." appears in the filename, and the filename is less than nine characters long.

Large random files are supported. The maximum logical record number is 32767. If a record size of 256 bytes is specified, then files up to eight megabytes can be accessed.

To open a file, you should understand the differences between a random file and a sequential file, which are summarized in the following paragraphs and covered in detail in Chapter 6, Page 6.1, "File Handling".

Sequential Files

A BASIC program can create and access two types of disk data files: sequential files and random access files. In sequential files, the data that is written onto the disk is stored one item after the other, in the same order it is sent. It is then read back in the same order. Thus, there are limitations in terms of speed and flexiblity because BASIC has to read through all the data sequentially, whenever the file is accessed.

One advantage to using sequential files is that there are fewer program steps involved in opening, reading, or writing a sequential file. Another advantage is, that generally, sequential files require less "overhead" space than random files.

PROGRAM ENTRY

Files and File Naming

Random Files

Random files are stored on the disk in packed binary formats and accessed in distinct units called records. Each record is numbered, thus allowing the data to be accessed randomly. Because the data can be accessed anywhere on the disk, it is not necessary to read through all the information, as with sequential files.

For further information on creating and accessing data files, see "File Handling," Chapter 6, Page 6.1.

PROGRAM ENTRY

Control Characters

BRIEF

Format: `CTRL {}`

Control characters are keyboard entries that affect the performance of your terminal and/or the output of the program being executed.

To execute any of the following control characters, you must hold down the control (CTRL) key and press the appropriate letter.

Details

The following control characters are used by Z-BASIC:

CTRL-C	Interrupts program execution and returns to the BASIC command level.
CTRL-G	Rings the bell at the terminal.
CTRL-H	BACK SPACE. Deletes the last character typed.
CTRL-I	TAB. Tab stops are every eight columns.
CTRL-J	LINE FEED. Subsequent text starts on the next line without entering a RETURN.
CTRL-S	Suspends program execution. Any key resumes program execution after a CTRL-S.
CTRL-U	Deletes the line that is currently being typed.

PROGRAM ENTRY

Syntax Notation

BRIEF

The following notation is used throughout this manual in descriptions of command and statement syntax. The syntax diagrams in the Alphabetical Reference Guide are labeled "Format".

Details

[ ]	Square brackets indicate that the enclosed entry is optional.
< >	Angle brackets indicate user-entered data. When the angle brackets enclose lower-case text, the user must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose upper-case text, the user must press the key named by the text; for example, <CTRL>.
{ }	Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.
. . .	Ellipses indicate that an entry may be repeated as many times as needed or desired.
CAPS	Capital letters indicate portions of the statements or commands that must be entered, exactly as shown.
¦	The stile indicates either/or. You must use the syntax on either the right or left side of the stile, but not both.

All other punctuation, such as commas, colons, slash marks, and equal signs must be entered exactly as shown.

Delimiters Used in Z-BASIC Printing

BRIEF

Delimiters separate items by marking their ends (limits). Many different delimiters are used at all levels of the computer system to mark the beginning and ending of things and to separate items in a series.

The comma is used to print separate expressions in fixed evenly-spaced locations on the line. The semicolon prints expressions in non-tabular format, placing the expressions at short, fixed distances without regard to how they line up.

Details

Often the words delimiter and terminator are used interchangeably. In this section, delimiter will be discussed in relation to printing/formatting techniques. In other words, *delimiters* are used to separate two adjacent expressions, whereas terminators, (discussed in Chapter 6), mark the end of items of data, including certain conditions that terminate data.

THE COMMA

Printing Tabular Formatted Data

The comma is used in PRINT statements to separate expressions, and it causes them to be printed at fixed, evenly-spaced locations along the line. This is a very useful technique for printing out tabular data. It is also very useful for printing out several variables with one PRINT statement.

Enter the following characters and observe the results:

```
A = 1111
B = 2222
C = 3333
D = 4444

Ok
PRINT A,B,C,D
 1111          2222             3333            4444
Ok

PRINT -D,-C,-B,-A
-4444         -3333            -2222           -1111
Ok
```

Delimiters Used in Z-BASIC Printing

The exact behavior of the comma in a PRINT statement depends on the structure of the output line. Each line of print is divided into a certain number of print zones or fields. The Z-100 has five print zones with 14 characters per field. Therefore if there are more than five values, they will be printed on more than one line.

As you can see from the example, on Page 2.19, the comma instructs the interpreter to print the next number beginning at the left edge of the next print field. The output is said to be left-justified within each field. It is important to remember if the number is positive the plus sign is not printed. A blank is printed in front of all positive numbers.

THE SEMICOLON

Items delimited by the semicolon are not printed in a tabular format (unless they all happen to be the same length). Instead, the interpreter prints numbers separated by a semicolon a short, fixed space apart from one another, without regard for how they line up with values above or below. This is valuable for getting the maximum number of output values on a line, as shown by the following example:

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

versus

```
PRINT 1,2,3,4,5,6,7,8,9,10
 1             2             3             4             5
 6             7             8             9             10
Ok.
```

CHAPTER 3 EDITING Z-BASIC PROGRAMS

The Full Screen Editor

BRIEF

The Z-BASIC full screen editor makes it possible to edit program lines anywhere on the screen.

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

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

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 later in this chapter.

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.

Details

The time saving benefit of the full screen editor during program development cannot be over-emphasized. We suggest you enter a sample program and practice each edit command until it becomes second nature.

Cursor

In the following discussion of edit commands, the term *cursor* refers to the marker (it can be blinking, reverse video, a block, or an underline) that indicates the current position on the screen.

The ability to edit anywhere on the screen makes it difficult to provide clear examples of command usage in printed text. The best way to get a "feel" for the editing process is to try editing a few lines while you study the edit commands that follow.

EDITING Z-BASIC PROGRAMS

The Full Screen Editor

THE EDIT COMMAND

With the full screen editor, the EDIT command simply displays the line specified and positions the cursor under the first digit of the line number. You can then modify the line by using the keys described in this chapter.

The format of the EDIT command is:

```
EDIT <line number>
EDIT.
```

Line number

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 command always gets the last line referenced by an EDIT command, 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.

INPUTTING Z-BASIC PROGRAMS

Any line of text you type while BASIC is in the direct mode will be processed by the full screen editor. BASIC is always in direct mode after the prompt `Ok`.

Any line of text you type that begins with a numeric character is considered a *program statement* and will be processed in one of six ways:

1. A new line is added to the program. This occurs if the line number is legal (range is 0 through 65529) and at least one non-blank character follows the line number in the line.

2. An existing line is modified. This occurs if the line number matches the line number of an existing line in the program. This line is replaced with text of the newly entered line.

3. An existing line is deleted. This occurs if the line number matches the line number of an existing line and the entered line contains *only a line number*.

4. An error is produced.

5. If you attempt to delete a non-existent line, an `Undefined line number` error message is displayed.

6. If program memory is exhausted, and a line is added to the program, the error `Out of Memory` is displayed and the line is not added.

You may place more than one BASIC statement on a line, but, separate each statement on a line from the last with a colon (:).

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

It is possible to extend a logical line over more than one physical full screen by using the line feed key, (CTRL-J). Typing a line feed causes subsequent text to start on the next full screen without entering a RETURN. When you finally enter a RETURN, the entire logical line is passed to BASIC for storage in the program.

Occasionally, BASIC may return to the direct mode with the cursor positioned on a line containing a message issued by BASIC such as `Ok`. When this happens, BASIC automatically erases the line. If the line were not erased and you typed a RETURN, the message would be given to BASIC and a `Syntax Error` would result. BASIC messages are internally terminated by HEX 'FF' to distinguish them from user text. This, however, is transparent to you.

EDITING Z-BASIC PROGRAMS

The Full Screen Editor

CHANGING A Z-BASIC PROGRAM

You can modify existing programs by displaying program lines on the screen with the LIST statement. You should first list the range of lines to be edited, (see the LIST statement in the Alphabetical Reference Guide). Then, position the cursor on the line to be edited, modify the line using the keys described in this chapter. Then type **RETURN** to store the modified line in the program.

NOTE: A program line is not actually modified within the BASIC program until RETURN is pressed. Therefore, when several lines need alteration, it is sometimes easier to move around the screen making corrections to several lines at once, and then, go back to the first line changed and press **RETURN** at the beginning of each line. By doing so, you will store the modified line in the program.

It is not necessary to move the cursor to the end of the logical line before typing the RETURN. The full screen editor remembers where each logical line ends and transfers the whole line even if the RETURN is typed at the beginning of the line.

To truncate a line at the current cursor position, type **CTRL-E**, followed by a **RETURN.**

SYNTAX ERRORS

When a syntax error is encountered during program execution, Z-BASIC automatically enters EDIT at the line that caused the error. For example:

```
10 A=2$12
RUN
Syntax error in 10
Ok
10 A=2$12
```

The full screen editor has displayed the line in error and positioned the cursor under the digit 1. To correct this error you would move the cursor right to the dollar sign ($) and change it to a caret ^ , followed by a RETURN. The corrected line is now stored in the program.

In this example, storing the line in the program causes all variables to be lost. If you wanted to examine the contents of some variable before making the change, you would type **CTRL-C** to return to the direct mode. The variables would be preserved since no program line was changed, and after you are satisfied, you can then edit the line and rerun the program.

LOGICAL LINE DEFINITION AND INPUT STATEMENTS

In the direct mode, a logical line always consists of all of the characters on each of the physical lines which make up the logical line. However, during the execution of an INPUT or LINE INPUT statement, this definition is modified slightly in order to allow for "forms" input. The logical line is restricted to characters actually typed or passed over by cursor movement.

I CHR and DELETE only move characters within the logical line. DELETE will decrease the size of the logical line. I CHR increases the logical line *except* when the characters moved will write over non-blank characters on the end of the logical line.

EDITING Z-BASIC PROGRAMS

The Full Screen Editor — Key Assignments

BRIEF

The full screen editor uses special keys and special key combinations to perform the following tasks: moving the cursor, inserting text, and deleting text.

The keys used to move the cursor to a location on the screen are:

- Cursor up
- Cursor down
- Cursor left
- Cursor right
- HOME key
- TAB key (with insert off)

The keys used to insert text are:

- I CHR (Insert Mode Toggle)
- CTRL-R

The keys used to delete text are:

- DELETE key
- BACKSPACE key
- CTRL-U (erase line)
- CTRL-E (erase to end of line)
- CTRL-L (clears the screen)
- CTRL-Z (erase to end of page)
- D CHR

The Full Screen Editor — Key Assignments

Details

The full screen editor recognizes the cursor movement keys located on the numeric keypad, the back space key, the ESC key, in addition to special key combinations for moving the cursor to a location on the screen, inserting characters, or deleting characters. The keys and their values are found in Table 3.1.

HEX	DEC	KEY	Function
15	21	CTRL-U	erase line
05	05	CTRL-E	erase EOL(end of line)
OC	12	CTRL-L	erase page
1A	26	CTRL-Z	erase EOP(end of page)
0B	11	HOME	position cursor in upper left-hand corner
15	21	F0	same as CTRL-U
1E	30	↑	cursor-up
1F	31	↓	cursor-down
1C	28	→	cursor-right
1D	29	←	cursor-left
12	18	I CHR	enter insert mode
7F	127	D CHR	delete character
03	03	CTRL-C	break
0E	14	CTRL-N	move to EOL
06	06	CTRL-F	forward word
02	02	CTRL-B	back word
12	18	CTRL-R	same as I CHR (insert character)
7F	127	DELETE	same as D CHR (delete character)
17	23	CTRL-W	delete word right of cursor
8	8	BACKSPACE	deletes last character typed

TABLE 3.1
Full Screen Editor Key Values

EDITING Z-BASIC PROGRAMS

The Full Screen Editor — Key Assignments

Description	Key
Moves the cursor to the upper left hand corner of the screen.	**HOME**
Clears the screen and positions the cursor in the upper left-hand corner of the screen.	**CTRL-L**
Up arrow. Moves the cursor up one line.	**CURSOR UP**
Down arrow. Moves the cursor one position down.	**CURSOR DOWN**
Left-pointing arrow. Moves the cursor one position left. When the cursor is advanced beyond the left of the screen, it will be moved to the right side of the screen on the preceding line. If it is on the top line, it will stop at the left corner.	**CURSOR LEFT**
Right-pointing arrow. Moves the cursor one position right. When the cursor is advanced beyond the right of the screen, it will be moved to the left side of the screen on the next line down. If it is on the bottom line, it will stop at the right corner.	**CURSOR RIGHT**
Depressing the CTRL and F key moves the cursor right to the next word. The next word is defined as the next character after an intervening blank to the *right* of the cursor in the set A..Z or 0..9.	**CTRL-F**
Depressing the CTRL and B keys moves the cursor left to the previous word. The previous word is defined as the next character after an intervening blank to the *left* of the cursor in the set A..Z or 0..9.	**CTRL-B**

EDITING Z-BASIC PROGRAMS

The Full Screen Editor — Key Assignments

CTRL-N Depressing the CTRL and N key moves the cursor to the end of the logical line. Characters typed from this position are appended to the line.

CTRL-E Depressing the CTRL and E key erases to the end of logical line from the current cursor position. All physical screen lines are erased until the terminating RETURN is found.

I CHR or CTRL-R Toggles insert mode. If Insert Mode is off, turns it on. If on, then turns it off.

When the insert mode is *off*, characters typed will replace existing characters on the line.

When the insert mode is *on*, characters following the cursor are moved to the right as typed characters are inserted at the current cursor position. After each keystroke, the cursor moves one position to the right. Line folding is observed. As characters advance off the right side of the screen they are inserted from the left on subsequent lines.

When the insert mode is *off*, depressing the TAB key moves the cursor over characters until the next tab stop is reached. Tab stops occur every eight character positions.

When the insert mode is *on*, depressing the TAB key causes blanks to be inserted from the current cursor position to the next tab stop. Line folding is also observed.

DELETE or D CHR Deletes one character immediately to the right of the cursor for each key depression. All characters to the right of the character deleted are moved one position to the left to fill in the character deleted. If a logical line extends beyond one physical line, characters on subsequent lines are moved left one position to fill in the previous space. The character in the first column of each subsequent line is moved up to the end of the preceding line.

EDITING Z-BASIC PROGRAMS

The Full Screen Editor — Key Assignments

BACK SPACE Causes the last character typed to be deleted, or deletes the character to the left of the cursor. All characters to the right of the cursor are moved to the left, one position. Subsequent characters and lines within the current logical line are moved up as with the DELETE key.

F0 or CTRL-U When typed anywhere in the line, it erases the entire logical line.

CTRL-C Returns to the direct mode, *without* saving any changes that were made to the current line being edited.

CTRL–W Deletes the next word.

CTRL-Z Erases to the end of the page.

The following figures illustrate the Z-100 keyboard. For more, detailed, information on the keyboard, refer to the Z-100 User's Manual.

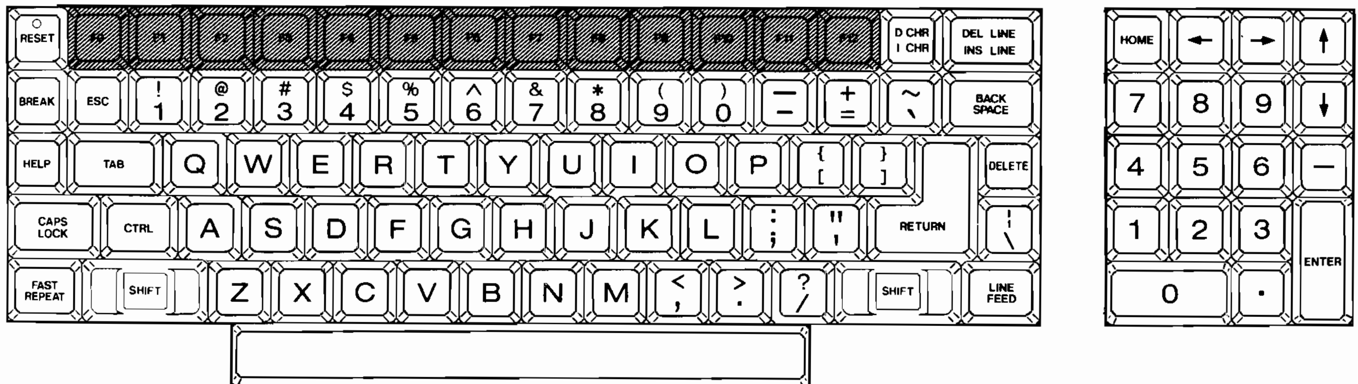


FIGURE 3.1
Function Keys

EDITING Z-BASIC PROGRAMS

The Full Screen Editor — Key Assignments

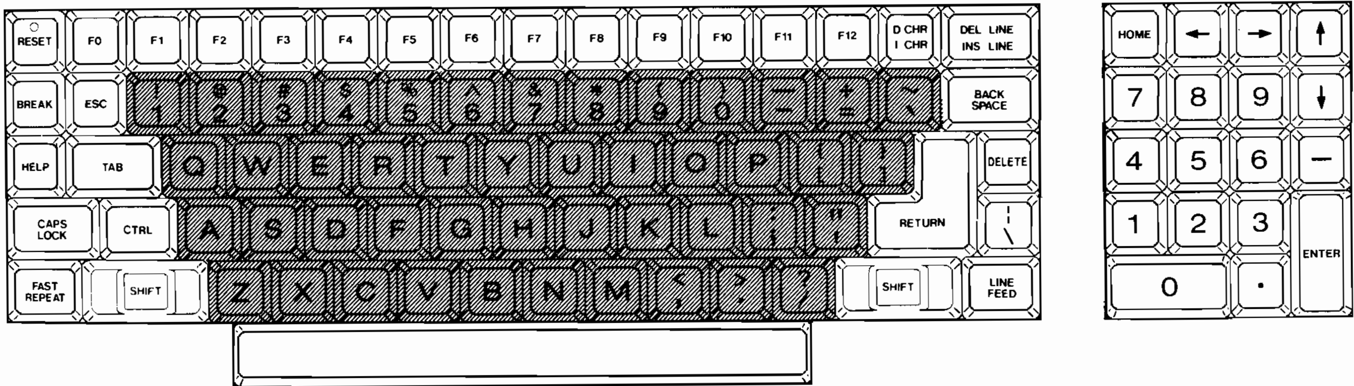


FIGURE 3.2
Alphanumeric Keys

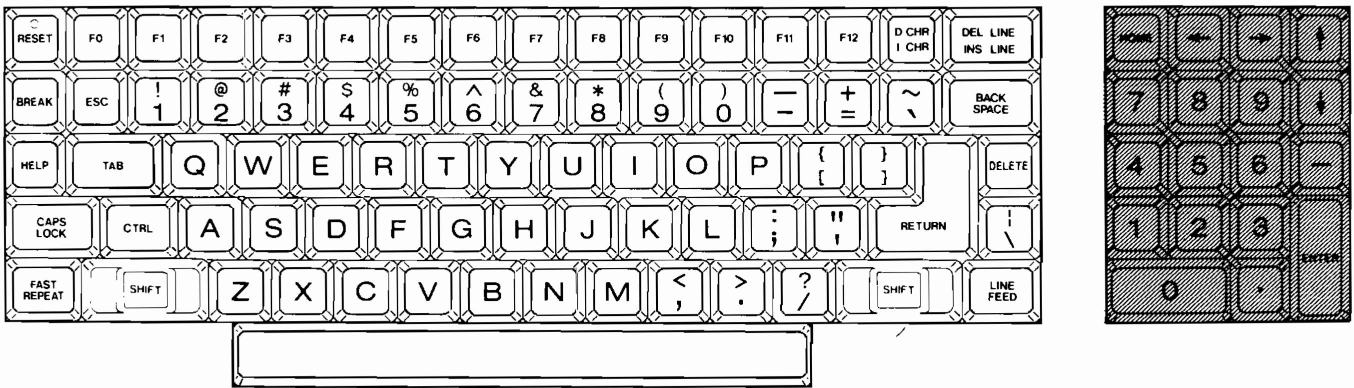


FIGURE 3.3
Keypad

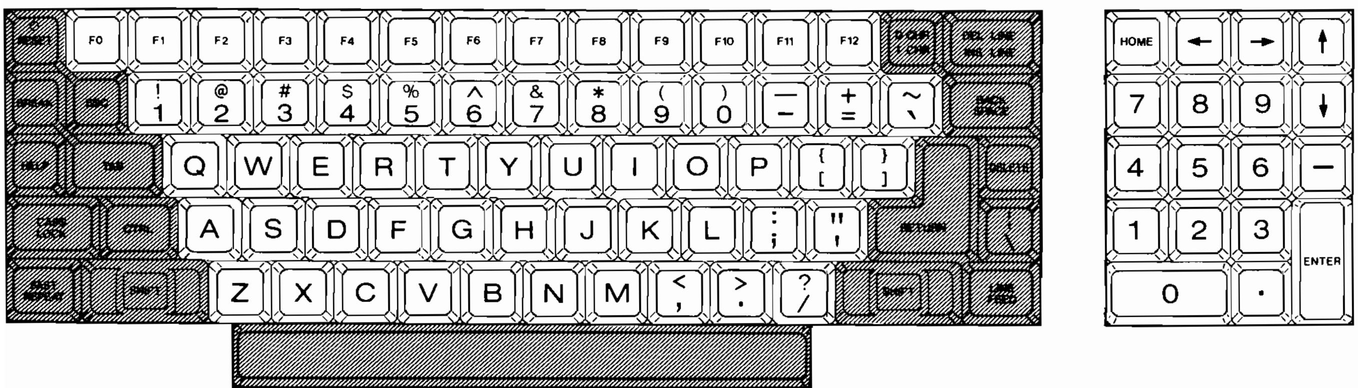


FIGURE 3.4
Special Keys

CHAPTER 4 PROGRAMMING IN Z-BASIC

Loading the BASIC Interpreter

BRIEF

This section will tell you how to operate Z-BASIC and explain the unique features of the Z-BASIC programming environment. No attempt will be made to teach the subject of BASIC programming, but enough information will be provided so that you should be able to gain some experience using the Z-BASIC Interpreter.

Details

The Z-BASIC Interpreter, which must be loaded into your computer's memory before you can use it, is an absolute binary file. This means that it is in a form that can be directly executed by your computer. Before you can perform the procedures listed below, you must "boot up" your computer. If you are unsure of how to do this, refer to your Z-DOS manual.

The Z-DOS filename used to reference the Z-BASIC Interpreter is Z BASIC. So, to load the Z-BASIC Interpreter into memory, type the following response to the prompt from Z-DOS:

A: **ZBASIC**

(Do not type the A:, as this represents the prompt from Z-DOS. Remember to terminate the line by pressing the **RETURN** key.)

This assumes that the file Z-BASIC resides on the current default drive. If the file does not reside on the current default drive, type the drive name and then the file name. For example, if A is the current default drive and the Z-BASIC file resides on drive B, you would use the following command to load Z-BASIC:

A: **B:ZBASIC**

PROGRAMMING IN Z-BASIC

Loading the BASIC Interpreter

After BASIC is loaded into memory, a sign-on message will be displayed on your screen. The amount of free memory, as well as the BASIC version number, will also be displayed (see "Starting Z-BASIC" Page 2.1). Take note of the amount of free memory, as this will no doubt be an important issue if you wish to write large, complex programs.

When BASIC is loaded in the manner described above, it will make certain assumptions about the operating environment. BASIC assumes that:

Workspace will be allocated dynamically
All available memory will be used,
The maximum number of files that can be open at one time is 255.

Filename

If <filename> (the file name of a BASIC program) is present, BASIC proceeds as if a RUN <filename> command were given after initialization is complete. A default file extension of .BAS is assumed if none is given. This allows BASIC programs to be batch run by putting this form of the command line in a Z-DOS AUTOEXEC.BAT file. Programs run in this manner will need to exit via the system in order to allow the next command from the AUTOEXEC.BAT file to be executed.

You can also specify the highest memory location BASIC will use with the /M: switch. In some cases it is desirable to set the amount of memory to allow reserved space for assembly language subroutines. If the /M: switch is omitted, all available memory will be used.

PROGRAMMING IN Z-BASIC

Loading the BASIC Interpreter

NOTE: The highest memory location number can be either decimal, octal (preceded by &O), or hexadecimal (preceded by &H).

Examples:

```
A:ZBASIC PAYROLL.BAS
```

Use all memory load and execute PAYROLL.BAS

```
A:ZBASIC /M:32768
```

Use first 32K of memory.

After the BASIC Interpreter has been loaded into memory, a program may be written.

PROGRAMMING IN Z-BASIC

Writing a BASIC Program

A BASIC program is composed of lines of statements containing instructions to BASIC. Each of these program lines begins with a line number (in the Indirect Mode), followed by one or more BASIC program statements. These line numbers indicate the sequence of statement execution, although this sequence may be changed by certain statements.

The format of a BASIC program line is:

Program Line Format

line number	**statement keyword**	**statement text**	**line terminator**
100	LET	X = X+1	RETURN

(space) between line number and statement keyword; (space) between statement keyword and statement text.

Every program line constructed in the Indirect Mode must begin with a line number, which must be an integer within the range 0 – 65529. This BASIC line number is a label that distinguishes one line from another within a program. Thus, each line number in the program must be unique.

Each program line in a BASIC program is terminated with a RETURN.

PROGRAMMING IN Z-BASIC

Writing a BASIC Program

When numbering program lines, you could use consecutive line numbers like 1,2,3,4. For example:

```
1 X = 1
2 Y = 2
3 Z = X+Y
4 END
```

However, a useful practice is to write line numbers in increments of 10. This method will allow you to insert additional statements later between existing program lines.

```
10 X = 1
20 Y = 2
30 Z = X+Y
40 END
```

AUTO Command

Another useful practice is to let BASIC automatically generate line numbers for you. This is accomplished with the AUTO command. The AUTO command tells BASIC to automatically generate line numbers. For example, if you type **AUTO 100,10**, then BASIC will generate line numbers beginning with line number 100 and incrementing each line by 10. Then all you need to do is type the BASIC program line after the generated line number. For more information on using the AUTO command, see the Alphabetic Listing of Commands in the Reference Guide, Page 10.4 of this manual.

PROGRAMMING IN Z-BASIC

Running a BASIC Program

After a BASIC program has been written, the next step is to execute the program. This can be accomplished by the RUN command. The following command would tell BASIC to execute the program currently in memory:

RUN

Program Execution

Execution would begin at the lowest line number and continue with the next lowest number line (unless the sequence of execution was altered with a statement like the GOTO statement). The RUN command can also specify the first line number to be executed. For example, the following command would cause execution to begin with line number 100:

RUN 100

You can also use the RUN command to execute a BASIC program that is currently residing on a disk file. For example, assume the file ALBUM.BAS resides on the current default disk. The following command would be used to execute ALBUM.BAS:

RUN"ALBUM"

Note that no drive specification or file name extension was included in the file name string. In this case, the current default drive and the extension .BAS are assumed.

PROGRAMMING IN Z-BASIC

Debugging a BASIC Program

Syntax and Logical Errors

In some cases, a BASIC program will not execute as you expected. This is usually the result of either a syntax error or a logic error. A syntax error is much easier to detect because BASIC will not only detect syntax errors for you, but will also point out the offending program line and invoke the Edit Mode. A logic error is much harder to detect, but several error trapping statements have been provided to make this an easier task.

When BASIC detects a syntax error, it will automatically enter Edit Mode at the line that caused the error. The full screen editor will automatically list the line which caused the syntax error and place the cursor at the beginning of the program line. At this point you can use the full screen editor to correct the error.

Syntax errors are usually a result of a misspelled keyword or an incorrectly structured program line. Remember that BASIC requires all reserved words to be delimited by a space. The easiest way to correct a syntax error is to refer to the appropriate syntax diagram (format) in the reference guide.

Because of the interactive nature of BASIC, it is very convenient to debug a BASIC program. Several statements have been provided to help you debug a BASIC program. But your first step is to find out the nature of the "bug".

A program "bug" may cause the wrong values to be output, or it may cause a program to branch to the wrong statement. The results of a calculation may be wrong or incomprehensible. A program "bug" might cause an error condition to be flagged. Often you must discover what the program is doing at the time of an error before you can determine the problem.

Also keep in mind that, in most cases, it is a bug in your program that is causing a problem. It is highly unlikely that the BASIC Interpreter is at fault.

PROGRAMMING IN Z-BASIC

Debugging a BASIC Program

Trace Flag

Once you have decided what the program is doing, you can take steps to discover why it is not executing correctly. For example, assume that a program is branching to a line number that is different from where you want it to branch. The trace flag has been provided to trace the flow of a program. To enable the trace, the TRON statement is used, and to disable the trace, the TROFF statement is used.

The trace flag will print each line number as it is being executed. The line number will be enclosed in square brackets ([]). It is best to generate a hard copy listing of the program first so you can follow this listing while the trace is running.

Breakpoints

Another important technique you can use in debugging is to set breakpoints in a program. You can use the STOP statement to temporarily terminate program execution, and then enter commands to print the values of various variables. You can also assign new values to these variables. Then you can continue program execution with a CONT command or a Command Mode GOTO.

Although you can print and change the values assigned to variables, you must not change the BASIC program after you interrupt execution with a STOP statement. If you do change the program, all the previously stored variable values will be lost, and all open files will be closed.

PROGRAMMING IN Z-BASIC

Saving a BASIC Program

When you have completed a BASIC programming session, you will no doubt want to save a copy of your most current program on the disk. This is accomplished with the SAVE command. The general format of the SAVE command is:

```
SAVE"<filename>"
```

The <file name> must be a valid Z-DOS file name. If no device specification is given, the current default drive will be assumed. If no file name extension is given, the default extension of .BAS will be assumed. For example, if you wish to save a program called GAME.BAS, you could use the following command:

SAVE"C:GAME.BAS"

Options Available For Saving

Note that this file will be written on drive C. The file name extension of.BAS could have been omitted, and then it would have been supplied as the default. BASIC will usually save files in a compressed binary format. A program can optionally be saved in ASCII format, but it will take more disk space to store it this way. Saving a file in ASCII format will permit you to print the file on a line printer and also permits you to use the compiler if you desire to do so. To save a program in ASCII format, append an A to the end of the file name string. For example:

SAVE"C:GAME",A

This will save the file on drive C in ASCII format with a file name of GAME.BAS. You can also save a program in a protected format so it can not be listed or edited. Just append a P to the end of the file name string. For example:

SAVE"C:GAME",P

This file will be saved in an encoded binary format.

Warning: When this protected file is later run or (loaded), any attempt to LIST or EDIT this program will fail.

PROGRAMMING IN Z-BASIC

Loading a BASIC Program

When you begin a BASIC programming session, you may want to load a program from the disk into memory. This is accomplished with the LOAD command. The general form of the LOAD command is:

```
LOAD"<filename>"
```

For example, if you wanted to load the program PAYROLL.BAS, you could use the command:

```
LOAD "PAYROLL"
```

Note that the file name extension was omitted. BASIC will assume a file name extension of .BAS. Also note that the drive specification was omitted. In this case, the current default drive will be assumed.

You may specify the file name using capitals or lower-case letters. The BASIC interpreter will automatically convert the file name into capital letters. This applies to all string constants or variables that contain file names.

It is also possible to execute a program with the LOAD command. If you want to do this, append an R (for RUN) to the end of the file name string. For example:

```
LOAD "PAYROLL",R
```

This form of the LOAD command will load a program into memory and execute it as if a RUN command had been typed. All currently open files will remain open for use by the program.

Listing a BASIC Program to a Line Printer

At some point during your programming effort, you may want a hard copy listing of a BASIC program. A BASIC program is listed to a hard copy device in much the same manner as it is listed to a console device. Use the LLIST command.

The general form of the LLIST command is:

```
LLIST
```

This will list the current program on the hard copy device. It is also possible to specify the range of line numbers to be listed. For example, in order to list a single line, you can use the command:

LLIST 100

This will list only the line number 100. A range of line numbers can also be specified:

LLIST 100 – 500

This will list line numbers 100 through 500, inclusive. The LLIST command will direct the output to the Z-DOS LST: device. This logical device can be assigned to several different physical devices. Refer to your Z-DOS manual for information about this process.

Checkpoint

In summary, the process of creating a BASIC program usually consists of the following steps:

1. LOAD Z-BASIC
2. Enter program lines
3. Use RUN to execute the program
4. Debug the program
5. Save the program
6. List the program to the printer

To rerun the program at a later time, LOAD the program with the “R” option.

CHAPTER 5 ARITHMETIC AND STRING OPERATORS

Variables

BRIEF

Variables in BASIC are treated exactly as if they were the value that they represent. Variable names may not be any of the reserved words (see the list on Page 2.9). The names may be up to 40 characters.

Variables occur in two distinct types — numeric and string. String variable names are distinguished by a dollar sign ($) written as the last character. Numeric variables may be declared as: integers (2 bytes), distinguished by a percent sign (%); single-precision (4 bytes), distinguished by an exclamation point (!); or double-precision (8 bytes), distinguished by a number sign (#).

Both numeric and string variables may be used to define arrays. The maximum number of dimensions for a BASIC array is 255. The maximum number of elements per dimension is 32766.

Details

VARIABLE NAMES FOR NUMBERS AND FOR CHARACTER STRINGS

Numeric and string variables are names that are used for assigned values. A numeric variable always has a number as its value and a string variable always has a character or string of characters as its value. Variables are treated by BASIC in much the same way that constants are treated (see Page 5.48).

Variable Names

Names that you use for variables may consist of letters, numbers, and decimal points (or periods). In some instances, symbols that declare the type of the variable may be used as the last character of the variable name.

ARITHMETIC AND STRING OPERATORS

Variables

For example: "XA", "BILLING", "MARK1" and "QUAD12", are all valid names.

Variable Name Length

The names may be of any length from one to 40 characters. If you enter a variable name that is longer than 40 characters, a syntax error message will occur.

EXCEPTIONS TO NAMING VARIABLES

Numbers

Variable names must begin with a letter. Invalid names would be: "17PAGE", "1STONE" and "12MONTH7DAY".

Reserved Words

The names you give to variables may not be any of the reserved words (see "Reserved Words" on Page 2.9), but the names may contain imbedded reserved words. For instance, consider the following two examples:

`10 LOG = .000142`	This would be reported as an error, however,
`10 ANALOG = .000142`	This would be okay since "LOG" is only a part of the variable name.
	Likewise,
`10 ON$ = "Light On"`	Would cause an error,
	and
`10 ONL$ = "Light On"`	Would not cause an error.

No variable name should begin with FN because commands beginning with FN are assumed to be user-defined functions (See "DEF FN" on Page 10.33).

Symbols

No variable name should end with the symbols that are set aside specifically for the declaration of variable types unless that variable is intended to be of that specific type (these types are covered in the section on "Declaring Variable Types" on Page 5.7).

The symbols used for declaration of variable type may also be considered reserved because they are used for specific results in the use of variables. These symbols are:

% ! # $

COMMON USES FOR VARIABLES

Variables have many uses, but four of the most common uses are:

1. You want to process more than one value in the same manner.
2. You want to use the same value several times within the same program.
3. You want to reserve space.
4. You need to pass the values in one program to another program or want to retrieve values from a disk.

Examples of each of these four uses might be:

Processing Variables with Different Values

1. If you were calculating gross profit for each month, you would use the same formula, but the values would most likely change from month to month. Consider this formula:

Monthly gross profit = monthly sales
− monthly cost of goods sold.

ARITHMETIC AND STRING OPERATORS

Variables

You may want to use the variable name "Sales" as the monthly value of total sales, "Cost" as the monthly cost of the goods that you sold, and "Gross" as the result, which would be the value of your gross profits. You could then shorten the formula to:

Gross = Sales − Cost

"COST", "SALES" and "GROSS" are all considered numeric variables (note that they do not have a dollar sign, "$", as their last character).

You would then assign each month's values to the variable names in the formula. When you are using long or complicated formulas this assignment makes it very easy to process different values without rewriting the formula each time.

Repetition of a Variable in Several Locations with the same Value

2. If you want to write a program that creates a form letter to send out to your clients, you can use variables to make the letter seem personalized by repeating the name of the person that will receive the letter in several places.

   To do this, you may write the program so that it would insert your client's name everywhere you want it to appear in the letter. Your letter might be similar to the letter on the next page.

```
Dear (client's name):

We have some very interesting and startling
news that we would like to pass on to you
(client's name), that we think you will be
interested in hearing.

We are having our annual sale and we are
offering special discount rates to our good
customers like you, (client's name)...

Well, (client's name) that sums it all up. We
hope to hear from you soon.

                            Sincerely,

P.S. Don't forget (client's name),
      only ten more days. . . .
```

You could let the variable "N$" (pronounced "N-string") equal the value of the client's name in your program. Wherever "(client's name)" appears in the above letter, you could tell the program to use the value of N$. N$ is a string variable (as is designated by the ending dollar sign, "$").

Using Variables to Save Memory Space

3. If you were writing a program to solve for the area contained in various circular shapes, you could set a variable equal to a value that was several digits long. For example, you could allow the numeric variable "PI" to have the value of 3.141592653589887

   PI = 3.141592653589887

   The variable name "PI" is only two characters long. The value of PI which is 3.141592653689887 when written in double-precision, consists of 17 characters (counting the decimal). If the value of PI was needed in 20 separate places in your program, you would save approximately 440 characters by using a numeric variable.

ARITHMETIC AND STRING OPERATORS

Variables

Passing Values

4. If you needed a program to keep track of the names and addresses of your clients, you could use set up a variable (such as N$ in example 2) for the clients' names, a variable for their street addresses (perhaps A$), variables for their cities (C$), states (S$), zipcodes (Z$), and a variable for the clients' phone numbers (P$).

   When you have input all six of these client data for each client into the computer, you could store the data on a disk without typing each item of data again. The transfer of data from one location to another or from one variable name to another name is sometimes referred to as passing values.

   An example of this process in English would be to tell the computer:

   > Start with the value of variable n set to (1), where n is a variable that counts the number of times that the instructions have been repeated. For each of my 96 clients do the following instructions.
   >
   > Let the variable N$ equal the value of my nth client's name
   >
   > Let the variable A$ equal the value of my nth client's street address
   >
   > Let the variable C$ equal the value of my nth client's city
   >
   > Let the variable S$ equal the value of my nth client's state
   >
   > Let the variable Z$ equal the value of my nth client's zipcode
   >
   > Let the variable P$ equal the value of my nth client's phone number
   >
   > Write N$, A$, C$, S$, Z$ and P$ to disk
   >
   > Increment n by 1 (add one to the value of variable n)
   >
   > If n is equal to 97 then stop
   >
   > If n is less than 97 then return to the top of this instruction list and get the new nth client's data

ARITHMETIC AND STRING OPERATORS

Variables

This example would reduce the actual handling of each of the items of data by allowing you to use variables whose values you could change in each pass. For instance, on each repetition, the variable N$ (and all of the other variables) would be assigned a different value. Then the value that was assigned to N$ (along with the values for the other variables) would be written to the disk in the order defined by the line "Write N$, A$, C$, S$, Z$ and P$...."

The following is a sample program included to demonstrate how this program would appear when written in BASIC.

```
90 OPEN "DATAFILE" FOR OUTPUT AS #1
100 N=1
110 LINE INPUT "CLIENTS NAME: ";N$
120 LINE INPUT "ADDRESS: ";A$
130 LINE INPUT "CITY: ";C$
140 LINE INPUT "STATE: ";S$
150 LINE INPUT "ZIPCODE: ";Z$
160 LINE INPUT "PHONE: ";P$
170 WRITE #1,N$,A$,C$,S$,Z$,P$
180 N=N+1
190 IF N=97 THEN END
200 GOTO 110
```

DECLARING VARIABLE TYPES

You may assign a type to variable names by using a symbol at the end of that name. When you make this assignment, you are said to be "declaring" that variable's type. There are two types of variables that have been mentioned so far: string and numeric. Numeric variables also may be declared to be of a specific precision.

ARITHMETIC AND STRING OPERATORS

Variables

Declaring String Variables

Declare string variables by using a dollar sign ($) as the last character of the variable name.

Example:

```
EXAMPLE$ = "This is a literal expression"
```

In the above example, "EXAMPLE" is the variable name, "$" declares that the variable name is a string variable, and "This is a literal expression" is the value that has been assigned the name "EXAMPLE$". The dollar sign ($) tells BASIC that the variable name will be used to represent a string literal.

Declaring Numeric Variable Precision

Numeric variables' names may be declared as integer, single or double-precision. This tells BASIC how precisely it should retain the value you have assigned to a numeric variable name.

A computation is more precise and accurate when you are using a variable declared as double-precision. However, there are many instances when it is better to use less precision. Here are some of the reasons that less precision might be more desirable:

- Higher precision variables occupy more storage space. If memory space or disk space is critical to an application but high precision is not, it is wise to declare variables to have less precision so that they do not take up as much room.

- Higher precision numbers take the computer more time to manipulate in an arithmetic operation. If the speed of a program that must do several calculations is critical but precision is not, declaring variables to have less precision will allow the program to run faster.

Declaring a variable type to be of a specific precision will round off the value in a known manner if that value exceeds the limitations placed on it by the precision that is declared. Certain applications you want to use may require specific limitations to the numeric values that are used by equations. Declaring a variable's precision can ensure that a value will be within the specified limitations. The limitations for each of the precision types are covered below.

Declaring Integer Variables

Declare integer variables by using a percent sign (%) as the last character of the variable name.

Example:

```
A% = 2.736
```

would cause the value of A% to be 3

```
C% = -99.341
```

would cause the value of C% to be −99

```
INTEGER% = .87654321
```

would cause the value of INTEGER% to be 1

The declaration of a variable as an integer causes the variable's value to be rounded to the closest integer (whole number) if the value is not already an integer. In the case of a value of one-half (.5), the value is rounded up to the next higher integer. In the case of a negative one-half (−.5), the value is rounded down to the next lower integer. Examples of this would be:

```
VALUE% = 17.5
```

would cause the value of VALUE% to be 18

```
DECLINE% = -42.5
```

would cause the value of DECLINE% to be −43

```
RATE% = -.5
```

would cause the value of RATE% to be −1

A variable that is declared as an integer may not be set to a value that exceeds the range of −32768 to +32767, or BASIC will report an overflow.

ARITHMETIC AND STRING OPERATORS

Variables

Declaring Single Precision Variables

Declare single-precision variables by using an exclamation point (!) as the last character of the variable name.

```
Q! = 9876543210.0123456789
```

would cause the value of Q! to be 9.876544E+09

```
COUNT! = 123.456789
```

would cause the value of COUNT! to be 123.4568

```
CAR3! = -123456789
```

would cause the value of CAR3! to be -1.234568E+08

A variable that is declared as single-precision that exceeds seven digits is rounded to its closest value. Although the seventh digit is displayed, its accuracy is not dependable. See Pages 5.48–5.53

Declaring Double Precision Variables

Declare double-precision variables by using a number sign (#) as the last character of the variable name.

Example:

```
DEBIT# = 91283764518.28
```

would cause the value of DEBIT# to be 91283764518.28

```
WORTH# = 9988776655443322ll.9988776655443322ll
```

would cause the value of WORTH# to be 9.988776655443322D+17

```
DECIMAL# =.01234567890123456789
```

would cause the value of DECIMAL# to be 1.234567890123457D-02

A variable that is declared as double-precision that exceeds 16 digits is rounded to its closest value. Limitations apply to double-precision variables the same as they do to double-precision constants on Page 5.50.

On the next page, you will find a table that shows how the three precision declarations affect given values.

Original Value	Declared Integer	Declared Single Precision	Declared Double Precision
1234567890987654321	Overflow	1.234568E+18	1.234567890987654D+18
−1234567890987654321	Overflow	−1.234568E+18	−1.234567890987654D+18
9876543210.0123456789	Overflow	9.876544E+09	9876543210.012346
−9876543210.0123456789	Overflow	−9.876544E+09	−9876543210.012346
1234567890.0987654321	Overflow	1.234568E+09	1234567890.098765
−1234567890.0987654321	Overflow	−1.234568E+09	−1234567890.098765
987654.321	Overflow	987654.3	987654.321
−987654.321	Overflow	−987654.3	−987654.321
32769	Overflow	32769	32769
−32769	Overflow	−32769	−32769
32768	Overflow	32768	32768
−32768	−32768	−32768	−32768
987.654321	988	987.6543	987.654321
−987.654321	−988	−987.6543	−987.654321
299.5	300	299.5	299.5
−299.5	−300	−299.5	−299.5
123.4567890987654321	123	123.4568	123.4567890987654
−123.4567890987654321	−123	−123.4568	−123.4567890987654
.5	1	.5	.5
−.5	−1	−.5	−.5
.0987654321	0	9.876543E−02	.0987654321
−.0987654321	0	−9.876543E−02	−.0987654321

Table 5.1
Precision Declaration on Various Values

ARITHMETIC AND STRING OPERATORS

Array Variables

BRIEF

An array is an ordered list of data items. It can be a one-dimensional vertical array or a table of data items consisting of rows and columns.

These data items may be either string or numeric. Each one is referred to as an element.

Several sample routines have been provided which can be used to manipulate arrays. These sample routines can be used to add, multiply, transpose, and perform other useful operations on numeric arrays.

Details

ARRAY DECLARATOR

An *array* is an ordered list of data items that may be a one-dimensional vertical list or a table of data items consisting of rows and columns. Before an array is referenced, it should be "declared" by use of an array declarator. The *DIM statement* is used to declare and establish the maximum number of elements in an array. The general form of the DIM statement is:

```
DIM <name>[(<integer expression>)]
```

where:

<name> is a valid BASIC symbolic name.

DIM Statement

The <integer expression> is any valid integer expression which, when evaluated, will be rounded to a positive integer value. This positive integer value will then become the maximum number of elements associated with that specific array name. The maximum number of dimensions is 255. The maximum number of elements per dimension is 32766.

An array can also be declared without the use of the array declarator. When BASIC encounters a subscripted variable that has not been defined with a DIM statement, it will assume a maximum subscript of 10. Thus, an array can be established without the use of the DIM statement.

ARRAY SUBSCRIPT

Each element of an array is referenced by an array subscript appended to the end of the array name. This array subscript is an integer expression which references a unique element of the array. Consider the following examples:

```
A(1),D$(I,J,K)
Q1(2)
Z#(55)
```

Subscript Errors

Any attempt to reference an array element with a subscript that is negative will result in an `Illegal Function Call` error message. References to subscripts which are larger than the maximum value established by a DIM statement and references which contain too many or too few subscripts will generate a `Subscript Out of Range` error message.

OPTION BASE STATEMENT

Changing the Defaults

The minimum subscript for an array element is assumed to be 0. The array declarator A(10) actually establishes an 11-element array, A(0) – A(10). The OPTION BASE statement can be used to establish the minimum array subscript value as 0 or 1. The default value is zero. The following example illustrates the use of the OPTION BASE statement.

```
OPTION BASE 1
DIM A (10)
```

Duplicate Definition Error

This program segment will establish a 10-element array, A(1) – A(10). The OPTION BASE statement must appear before any DIM statement or before any subscripted variable is referenced. An attempt to use the OPTION BASE statement after an array has already been established will result in a `Duplicate Definition` error message . This same error message will occur if you declare the same array later in the same program without erasing the previous declaration of the array.

ARITHMETIC AND STRING OPERATORS

Array Variables

VERTICAL ARRAYS

Storage Allocation

A vertical array is a 1-dimensional array. You can establish this type of array by using the DIM statement or by letting BASIC establish the default array size. Assuming that the default array size of 11 elements has been established for the array A, Z-BASIC would allocate storage as follows:

Array element	Subscribed variable
Element #1	A(0)
Element #2	A(1)
Element #3	A(2)
Element #4	A(3)
Element #5	A(4)
Element #6	A(5)
Element #7	A(6)
Element #8	A(7)
Element #9	A(8)
Element #10	A(9)
Element #11	A(10)

Table 5.2
Array Storage Allocation

The variable A(9) would reference the tenth element of this vertical array. (Although the OPTION BASE statement could be used to set the minimum subscript to 1. In this case A(9) would reference the ninth element of the array.)

MULTI–DIMENSIONAL ARRAYS

A multi-dimensional array is declared in the same manner as a vertical array, except that both row and column size are declared. For example, to declare a 3 × 3 array, the following sequence of statements could be used:

```
OPTION BASE 1
DIM A(3,3)
```

After this program segment is executed, BASIC would reserve nine storage locations for the array. (Note that the minimum subscript value was set to 1 with the OPTION BASE statement.)

Column	1	2	3
Row 1	A(1,1)	A(1,2)	A(1,3)
2	A(2,1)	A(2,2)	A(2,3)
3	A(3,1)	A(3,2)	A(3,3)

Table 5.3
Multi-Dimensional Array Storage Allocation

When you are reading from left to right, note that the second array subscript varies most rapidly.

String Arrays

String arrays can also be established in the same manner as numeric arrays. A string array is declared when the DIM statement is used.

```
DIM A$(100)
```

This statement will establish a 101-element string array. To access an element of the array, append an array subscript to the end of the variable name.

```
A$(20)="A STRING ARRAY"
```

MATRIX MANIPULATION

Matrix Input Subroutines

A collection of subroutines that are very useful for manipulating a matrix are shown below. The subroutine line numbers in the following example may have to be changed to be compatible with your program.

```
5000 'SUBROUTINE NAME -- MATIN2
5010 'ENTRY I% = # OF ROWS, J% = # OF COLUMNS
5020 DIM MAT(I%,J%)
5030 FOR K% = 1 TO I%
5040 PRINT "INPUT ROW #";K%
5050        FOR L% = 1 TO J%
5060        INPUT MAT(K%,L%)
5070 NEXT L%,K%
5080 RETURN
```

The above subroutine will accept data from the terminal and assign this data to the 2-dimensional array named MAT. Upon entry into this subroutine, the integer variable I% must contain the number of rows in the matrix and J% must contain the number of columns.

```
5000 'SUBROUTINE NAME — MATIN3
5010 'ENTRY         I% = SIZE OF DIMENSION #1
5020 '              J% = SIZE OF DIMENSION #2
5030 '              K% = SIZE OF DIMENSION #3
5040 DIM MAT (I%,J%,K%)
5050 FOR L% = 1 TO I%
5060       FOR M% = 1 TO J%
5070                 FOR N% = 1 TO K%
5080 READ MAT(L%,M%,N%)
5090 NEXT N%,M%,L%
6000 RETURN
```

This subroutine listed above is used to read data from DATA statements and assign this data to the 3-dimensional array named MAT. Upon entry into this subroutine, the integer variable I% must contain the number of elements for dimension 1, J% must contain the number of elements for dimension 2, and K% must contain the number of elements for dimension 3. Also, the data must be contained in valid DATA statements.

SCALAR MULTIPLICATION

Multiplication by a Single Variable

```
5000 'SUBROUTINE NAME -- MATSCALE
5010 'ENTRY         I% = SIZE OF DIMENSION #1
5020 '              J% = SIZE OF DIMENSION #2
5030 '              K% = SIZE OF DIMENSION #3
5040 '       A--ORIGINAL ARRAY
5050 '   X--SCALAR FACTOR
5060 '   B--NEW ARRAY
5070 FOR L% = 1 TO K%
5080   FOR M% = 1 TO J%
5090     FOR N% = 1 TO I%
6000     B(N%,M%,L%) = A(N%,M%,L%)*X
6010     NEXT N%
6020   NEXT M%
6030 NEXT L%
6040 RETURN
```

This subroutine will multiply each element in the 3-dimensional array A by the value assigned to X and produce a new 3-dimensional array B. Upon entry into this subroutine, I% must contain the size of dimension #1,J% must contain the size of dimension #2,K% must contain the size of dimension #3, and X must be assigned the value to multiply by (scalar factor). Both arrays A and B must also have previously been defined by a DIM statement.

TRANSPOSITION OF A MATRIX

```
5000 'SUBROUTINE NAME -- MATTRANS
5010 'ENTRY I% = # OF ROWS, J% = # OF COLUMNS
5020 'TRANSPOSE A INTO B
5030 FOR K% = 1 TO I%
5040   FOR L% = 1TOJ%
5050   B(L%,K%) = A(K%,L%)
5060   NEXT L%
5070  NEXT K%
5080 RETURN
```

This subroutine will transpose the 2-dimensional matrix A into the 2-dimensional matrix B. Upon entry into the subroutine, I% must contain the number of rows in A and J% must contain the number of columns in A. The arrays A and B both must have previously been defined by a DIM statement.

ARITHMETIC AND STRING OPERATORS

Array Variables

Matrix Addition

```
5000 'SUBROUTINE NAME -- MATADD
5010 'ENTRY -- I% = SIZE OF DIMENSION #1
5020 '         J% = SIZE OF DIMENSION #2
5030 '         K% = SIZE OF DIMENSION #3
5040 'ARRAY A+B = C
5050 FORL% = 1 TO K%
5060   FORM% = 1 TO J%
5070     FOR N% = 1 TO I%
5080     C(N%,M%,L%) = B(N%,M%,L%) + A(N%,M%,L%)
5090     NEXT N%
6000    NEXT M%
6010   NEXT L%
6020 RETURN
```

This subroutine will add the elements of arrays A and B to produce a new array C. A, B, and C must have previously been defined by a DIM statement.

Matrix Multiplication

```
5000 'SUBROUTINE NAME -- MATMULT
5010 'ENTRY -- ARRAY A MUST BE D1% BY D3% ARRAY
5020 '         ARRAY B MUST BE D3% BY D2% ARRAY
5030 '         ARRAY C MUST BE D1% BY D2% ARRAY
5040 FOR I% = 1 TO D1%
5050   FOR J% = 1 TO D2%
5060   C(I%,J%) = 0
5070      FOR K%=1 TO D3%
5080      C(I%,J%)=C(I%,J%)+A(I%,K%)*B(K%,J%)
5090     NEXT K%
6000    NEXT J%
6010   NEXT I%
6020 RETURN
```

This subroutine will multiply the 2-dimensional array A by the 2-dimensional array B and produce C.

Using array variables is an advanced programming technique. If you are having problems understanding the preceding information, refer to other BASIC programming resources. See the bibliography at the end of this manual.

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

ARITHMETIC OPERATORS

BRIEF

The arithmetic operators in BASIC are the symbols +, −, /, \, MOD, *, and ^, which stand for addition, subtraction, division, integer division, modulo arithmetic, multiplication, and exponentiation, respectively.

Integer Division, denoted by a (\) backslash, is an operator that rounds the operands to the nearest integer within the range of −32768 to 32767. The operands are rounded before division is performed, and the answer is rounded to a whole number as well.

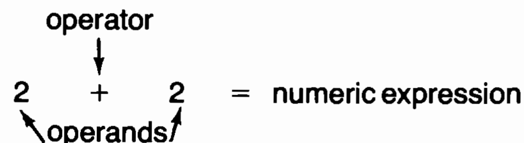
Modulus Division, denoted by the operator MOD, gives the remainder of the value that is the result of integer division.

Rules of precedence determine the order in which operators are evaluated.

Parentheses can be used to change the order of evaluation by indicating which operations are to be performed first.

Details

The BASIC arithmetic operators perform common arithmetic operations such as addition, subtraction, negation, division, multiplication and exponentiation. A numeric expression is any collection of operators and operands that can be arithmetically evaluated to produce a single numeric result.



ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

Precedence is a predetermined order in which expressions are evaluated. The following table demonstrates the order of precedence of BASIC arithmetic operators. The order that the operators are listed in reflects the order that they would be evaluated in an expression.

Precedence

Operator	Operation	Example
^	Exponentiation	A ^ B
–	Negation	–B
* /	Multiplication and Floating-point Division	A*B, A/B
\	Integer Division	A\B
MOD	Modulus Division	A MOD B
+ –	Addition and Subtraction	A+B, A–B

Table 5.4
Order of Precedence

It is important to take note of the order of precedence when you are setting up numeric expressions because the order in which an expression is evaluated can greatly affect the result.

Example:

```
10 PRINT 8+4^2/8*2
RUN
 12
Ok
```

In this numeric expression, 4 ^ 2 (four raised to the power of two) is the first expression that is evaluated, with a result of 16. Since the 16/8 is left of the 8*2, the division is carried out next. The value 16 is divided by 8 and then multiplied by 2 with a result of 4. Then the 8 is added which makes 12 the final result.

If the order of precedence was disregarded and the 8, for example, was added to (4 ^ 2) first: 24 would be the value divided by 8, which would cause 3 to be multipled by 2, yielding an incorrect result of 6.

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

Exponentiation

Exponentiation is used to handle very large or very small numbers in an abbreviated form. Exponentiation means to raise the value of the numeral on the left of the operator to the power of the numeral on the right. All exponentiation is performed from left to right as it appears in the expression.

Negation

Negation, the minus sign (−), can function in two different ways. If it is between two numbers, it stands for the subtraction operation, as in PRINT 8−5; but if it is in front of a number, it serves to indicate a negative quantity, as in PRINT −5. In PRINT 8−5, the minus sign is called a binary operator, because it has two operands (the numbers on either side of it); while in PRINT −5, it is called a unary operator because it only has one operand (the number following it).

In BASIC, the unary minus is a real operator, not just a piece of the number it's attached to. The operation is called negation and is the equivalent of multiplying the number by −1. Thus, the statement PRINT (−1)*5 is equivalent to PRINT −5. Unary operation is also used to demonstrate the relationship between logical operators as described on Page 5.32.

Multiplications and Divisions

Multiplications and divisions are then evaluated by BASIC, going from left to right in the expression. Multiplication is denoted by the (*) asterisk, which must be included between quantities, unlike mathematics where the symbol can sometimes be omitted.

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

Floating Point Division

Floating Point Division is denoted by a slash (/) and is performed in the usual arithmetic manner. However, a backslash (\) indicates integer division, which rounds the numbers to integers before division takes place. The quotient is also truncated to an integer. The operands must be in the range −32768 to 32767.

Example:

10\4 = 2
25.68\6.99 = 3

Overflow and Division by zero

If a division by zero is encountered during the evaluation of an expression, the `Division by zero` error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the `Division by zero` error message is displayed, and execution continues.

Similarly, if overflow occurs, the `Overflow` error message is displayed, and execution continues.

Modulus Division

Modulus division is denoted by the operator MOD. It gives the integer value that is the remainder (also known as the modulo) of an integer division expression. The remainder is also expressed as an integer value.

Example:

```
Ok
10 LET A= 5 MOD 3
20 PRINT A
RUN
 2
Ok
```

The result is 2 because 5\3 is 1, with a remainder of 2.

Additions and Subtractions

Finally, all additions and subtractions are evaluated, going from left to right. Here are some sample algebraic expressions and their BASIC counterparts:

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

Sample Expressions

Algebraic Expression	BASIC Expression
$X+2Y$	X+Y*2
$X-\frac{Y}{Z}$	X-Y/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X ^ 2) ^ Y
X^{Y^Z}	X ^ (Y ^ Z)
$X(-Y)$	X*(-Y)

Table 5.5
Algebraic Expressions and Their BASIC Counterparts

Parentheses

Two consecutive operators must be separated by parentheses such as in the case X*(-Y).

Parentheses can be used to change the order of evaluation by indicating which operation is to be performed first.

Checkpoint

Here are three examples that use parentheses to change the predetermined order of evaluation. Before you go on to the next page, study these examples to see if you can determine how the interpreter will handle the expressions.

A. (8+4) ^ 2/(8*2)

B. 8+4 ^ (2/8*2)

C. 8+(((4 ^ 2)/8)*2)

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

	Expression	Explanation
A.	(8+4) ^ 2/(8*2)	Original expression.
	12 ^ 2/(8*2)	Left-most set of parentheses: 8+4 is 12.
	144/(8*2)	Exponentiation: 12 ^ 2 is 144.
	144/16	Next set of parentheses: (8*2) is 16.
	9	Division: 144/16 is 9.
B.	8+4 ^ (2/8*2)	Original expression.
	8+4 ^ (.25*2)	Expression in parentheses comes first; division on the left is performed, replacing 2/8 with .25.
	8+4 ^ .5	.25*2 is .5.
	8+2	4 ^ .5 is 2.
	10	8+2 is 10.
C.	8+(((4 ^ 2)/8)*2)	Original expression. Notice that nested parentheses are evaluated from the inside out.
	8+((16/8)*2)	Inner parentheses are evaluated first: (4 ^ 2) becomes 16.
	8+(2*2)	Next set of parentheses: (16/8) is 2.
	12	8+4 is 12.

As you can see from these examples, the location of the parentheses in a numeric expression affect the result of that expression. Understanding the rules of precedence and the rules regarding parentheses will help you obtain the desired results from your programs. However, if you write an expression improperly or ask the computer to do something it cannot do, you will get error messages (which are discussed on the following pages).

Syntax Errors

Syntax errors occur when the BASIC interpreter attempts to translate a statement that is improperly written. The interpreter will not translate the statement and will print out a syntax error message.

Example:

You enter:	**PRNT 2+2**
BASIC responds:	`Syntax error`

A syntax error occurred because PRINT was misspelled.

You enter:	**PRINT 32 – /4**
BASIC responds:	`Syntax error`

A syntax error occurred in this case because the minus to the right of the 32 must be the binary subtraction operator. The / is always binary, and you cannot have two adjacent binary operators.

You enter:	**?(2*(12–4*3)**
BASIC responds:	`Syntax error`

A syntax error resulted because a parenthesis was omitted. This is a very common mistake. There must always be an even number of parentheses, since each left parenthesis must face toward a corresponding right parenthesis. The question mark causes no problems because Z-BASIC accepts the ? as a shorthand notation of the PRINT statement. A corrected version of this expression would be as follows:

```
? (2*(12-4*3))  or  ? 2*(12-4*3)
```

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

Execution Errors

Another kind of error you may get is called an execution error. An execution error occurs when a properly written command tells the computer to do something it cannot do. Processing stops, and an execution error message is displayed. Below, we've included several examples of execution errors and the conditions that cause them.

`PRINT 2+`	Missing the other operand.
`PRINT 2/0`	Result is mathematically undefined.
`PRINT (-3)^.5`	Produces a value that cannot be represented in the number system used by the interpreter.
`PRINT 100^999`	Produces a number too large for the interpreter to handle. This is called an overflow condition.
`READ X (No data statement included)`	Tries to read a nonexistent item of data.

RELATIONAL OPERATORS

BRIEF

A relational operator tells the interpreter to evaluate and compare the two expressions on either side of the operators.

Relational operators must always stand between two valid expressions of the same type, either both numeric or both string.

Usually, the result of the comparison is used to make decisions about program flow.

The result of the comparison is either true or false, which is why relational operators are used to form the condition of conditional branches.

Details

Relational operators are symbols used to evaluate and compare two expressions. They stand between two valid expressions, either both numeric or both string. Following are the relational operators used in BASIC.

Operator	Relation	Example
=	Equal to	A=B
<	Less than	A<B
>	Greater than	A>B
<=	Less than or equal to	A<=B
>=	Greater than or equal to	A>=B
<>	Not equal to	A<>B

Table 5.6
Relational Operators

Each of the relational operators listed in Table 5.6 can have an opposite or negative meaning as shown in Table 5.7.

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

Negative Meanings

Positive Meaning	Operator	Negative Meaning
Equal to	=	Not less than and not greater than
Less than	<	Not greater than and not equal to
Greater than	>	Not less than and not equal to
Less than or equal to	<=	Not greater than
Greater than or equal to	>=	Not less than
Not equal to	<>	Not equal to

Table 5.7
Negative Meaning of Relational Operators

Negation

Additionally, expressions that contain relational operators can be written using a negated structure.

Positive Meaning	Operator	Negation	Negative Meaning
Equal to	=	<>	Not equal to
Less than	<	>=	Not less than
Greater than	>	<=	Not greater than
Less than or equal to	<=	>	Not less than and not equal to
Greater than or equal to	>=	<	Not greater than and not equal to
Not equal to	<>	=	Not less than and not greater than

Table 5.8
Negated Structure of Relational Operators

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

If you replace a relational operator with its negation, the statement "switches branches" or takes the opposite course of action. The result of the following line:

```
100 IF A=B THEN 500
```

will always be the exact opposite of the result of

```
100 IF A<>B THEN 500
```

If the first statement branches to 500, the second continues to the next line. Conversely, a branch in the second statement will cause the condition to fail in the first.

A relational operator is often replaced with its negation to save space on the program line.

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

Inaccuracies

Slight inaccuracies can introduce minute differences between expressions that are theoretically equal. This can cause occasional problems in conditional statements.

Example:

```
10 A=99
20 B=SQR (A)
30 C=SQR (A)
40 IF B*C=A THEN PRINT "GOOD COMPARISON" ELSE PRINT "NOT
   EQUAL"
```

This program tells BASIC to get the square root of 99, and assign that value to variable B. Then in line 30, the square root of 99 is assigned to variable C. Line 40 says if the square root of 99 multiplied by the square root of 99 is equal to 99, then print, "GOOD COMPARISON", if it is not equal to 99 then print "NOT EQUAL".

When BASIC computes the square root of 99, the result is not 9, it is actually 9.949874. When BASIC multiplies this number by itself, the result is 98.99999. The IF statement in line 40 will always be false, unless you build a slight margin of error into the comparison. To correct this problem, the following program line was added:

```
50 IF (B*C-A)<0.0001 THEN PRINT "GOOD COMPARISON" ELSE
   PRINT "NOT EQUAL"
```

to allow for a difference of up to .00001 between B*C and A, and still have them treated as "equal". Another way to avoid this problem is by using integers in your calculations.

Numeric Comparisons

A numeric comparison evaluates and compares the values of two numeric expressions. The result of the comparison of expressions can be either true (−1) or false (0).

Arithmetic operations are always performed first when arithmetic operators are combined with relational operators.

Example:

```
A+B<(C-1)/D
```

This statement will be true (−1) if the value of A+B is less than the value of C−1 divided by D.

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

```
PRINT 8<2; 8<12
 0 -1
Ok
```

In this example, the first result is false (0) because 8 is not less than 2; and the second result is true (– 1) because 8 is less than 12.

String Comparisons

String comparisons are made alphabetically. A string is considered less than another string if it comes before another string alphabetically. Lowercase letters are greater than capital letters. Capital letters are greater than numbers. Punctuation values are divided, with the symbols : ; < = > and ? greater than the numbers 0–9, and ! " # $ % & ' ( ) *+ - and . less than the numbers 0–9. See Appendix C for a complete list of ASCII codes and their equivalent values.

String comparisons are made by taking one character at a time from each string and comparing the ASCII code values. These values are compared and evaluated with relational operators. Each character is compared separately. If the ASCII codes are the same in both string expressions, then the strings are said to be equal. If the ASCII code is different, the string with the lower code is less than the string with the higher code.

If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Spaces on either side of either expression are also counted. All string constants used in comparison expressions must be enclosed in quotation marks.

Examples:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78" where B$ = "10/12/60"
```

Thus, string comparisons can be used to test string values or to alphabetize strings.

LOGICAL OPERATORS

BRIEF

Logical operators are used to connect two or more relations (expressions that contain relational operators) and return a true or false value, which is used to make decisions regarding program flow.

The logical operators in BASIC are: NOT, AND, OR, XOR, IMP, and EQV.

Like relational operators, logical operators are governed by rules of precedence, unless modified with parentheses.

Logical operators permit you to manipulate the value of a bit, which is a unit of data in binary notation.

Logical operators are used to perform Boolean operations, which are used to evaluate binary variables.

Details

Logical operators are used to connect two or more relations and return a true (−1) or false (0) value. These values can be evaluated to make decisions about program flow. Like relational operators, logical operators are most often used in conditional statements such as the IF...THEN...ELSE statement.

Example:

1. `IF D<200 AND F<4 THEN 80`

2. `IF I>10 OR K<0 THEN 50`

3. `IF NOT P THEN 100`

The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The result of a logical operation is determined as shown in Table 5.9. This table is commonly known as a *truth table*, which is an enumeration of all possible values of the operands and their corresponding results. The operators are listed in order of precedence.

NOT

X	NOT X
T	F
F	T

AND

X	Y	X AND Y
T	T	T
T	F	F
F	T	F
F	F	F

OR

X	Y	X OR Y
T	T	T
T	F	T
F	T	T
F	F	F

XOR

X	Y	X XOR Y
T	T	F
T	F	T
F	T	T
F	F	F

IMP

X	Y	X IMP Y
T	T	T
T	F	F
F	T	T
F	F	T

EQV

X	Y	X EQV Y
T	T	T
T	F	F
F	T	F
F	F	T

Table 5.9
Truth Table

NOT

The *NOT* operator is the logical complement operator. The role of the NOT operator is to negate a logical expression. NOT is the only logical operator that works with one operand. For example the following statement:

```
200 IF A=B THEN 500
```

is the logical complement of:

```
200 IF NOT A=B THEN 500
```

In another example:

```
300 IF A=B AND C=D THEN 500
```

is the logical complement of:

```
300 IF NOT (A=B AND C=D) THEN 500
```

In other words, the second statement will produce the opposite result of the first statement.

When two NOT operators are applied to the same expression, they cancel each other out, just as two minus signs cancel each other in arithmetic. Thus, an easier way to write a statement such as NOT(NOT A=B) is A=B.

Under some circumstances, it can be valuable to use an equivalent expression. Two statements are said to be *equivalent* if they produce identical results under all different conditions. The following table gives the rules for equivalent expressions, called De Morgan's Laws.

De Morgan's Laws

DE MORGAN'S LAWS

If ≡ stands for logical equivalence, and the letters P and Q represent two logical expressions, then:

1. NOT (P OR Q)≡(NOT P) AND (NOT Q)
2. NOT (P AND Q)≡(NOT P) OR (NOT Q)
3. P OR Q≡NOT((NOT P) AND (NOT Q))
4. P AND Q≡NOT ((NOT P) OR (NOT Q))

Table 5.10
De Morgan's Laws

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

For example, using DeMorgan's Law, the following program lines can be converted to a simpler form.

```
220 IF C$>="A" AND C$<= "Z" THEN 250
230 IF C$>="0" AND C$<= "9" THEN 250
240 SY =SY+1
```

Note that this segment determines whether C$ is a "symbol" (not an alphabetic or a numeric character), and if it is, adds one to the symbol counter SY.

Step 1. Use OR to combine lines 220 and 230.

```
220 IF (C$>="A" AND C$<= "Z")
    OR (C$>="0" AND C$<= "9") THEN 250
230 (deleted)
240 SY=SY+1
```

Step 2. Apply De Morgan's Law #1 (twice).

```
220 IF (NOT(C$<"A" OR C$> "Z"))
    OR (NOT(C$<"0" OR C$>"9")) THEN 250
240 SY=SY+1
```

Step 3. Negate the condition in line 220 to "switch branches", which puts SY=SY+1 on line 220.

```
220 IF NOT ((NOT(C$<"A" OR C$>"Z"))
    OR (NOT(C$<"0" OR C$>"9"))) THEN SY=SY+1
240 (deleted)
```

Step 4. Apply De Morgan's Law #4.

```
220 IF (C$<"A" OR C$>"Z") AND
    (C$<"0" OR C$>"9") THEN SY=SY+1
```

AND

AND is the conjunction operator which tells the interpreter to compare two expressions, bit by bit, and return a true value only if every pair of bits is equivalent. A bit is a single binary digit that is the smallest element in computer storage capability. If you look again at the truth table, AND is only true when both X and Y are true.

OR

OR is the disjunction operator that says either X or Y or both X and Y must be true in order for the result of the expression to be true. An OR operator returns a zero only when both X and Y are false.

XOR

XOR is the exclusive OR operator that means either X or Y can be true, but not both of them. If both X and Y are true, or both X and Y are false, the result will be false.

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

IMP

IMP is an operator that means if the truth value of X implies the truth value of Y, then the expression is true. The only time the result is false is when the first logical expression is true and the second is false. The X assertion could be false; but as long as the Y assertion is an implication of X, the result is true. Consider the following program.

```
10 PRINT "SELECT TEMPERATURE (HOT, WARM, COOL, FRIGID): ";
20 T$=INPUT$(1): PRINT T$:PRINT
25 IF T$="H" OR T$="W" THEN T=-1 ELSE T=0
30 PRINT "SELECT PRECIPITATION (NONE, RAIN, HAIL, SNOW): ";
40 P$=INPUT$(1):PRINT P$:PRINT:PRINT
45 IF P$="N" OR P$="R" THEN P=-1 ELSE P =0
50 IF T IMP P THEN PRINT "THAT SOUNDS LOGICAL"     ELSE PRINT
"THAT SOUNDS SILLY"
60 FOR Z=1 TO 800: NEXT Z: GOTO 10
```

In this program, if the temperature outside is either hot or warm, it is logical to assume that there could be no precipitation or it may be raining. If it is cool or frigid, it is logical to assume it may be hailing or snowing. You could lie and say it was cold outside on a day when it was really hot. Then, if you said it was snowing on a cold day, that would be a true and logical assertion based on the first assertion that it was cold. A false or 0 value would only be returned if the second statement is not implied by the first. The following table may help you understand how the IMP operator was used in this example.

Temperature	
HOT	-1
WARM	-1
COOL	0
FRIGID	0

Precipitation	
NONE	-1
RAIN	-1
HAIL	0
SNOW	0

X	Y	X IMP Y
T	T	T
T	F	F
F	T	T
F	F	T

Table 5.11
The IMP Operator

EQV The EQV operator denotes equivalence. As noted in our discussion of De Morgan's Laws, two logical expressions are said to be equivalent if they produce identical results under all different conditions. If both X and Y are true or if both X and Y are false, then the result is true.

Example:

```
NOT (X OR Y) EQV ((NOT X) AND (NOT Y))
```

Precedence

You should remember that logical operators are governed by a certain order of precedence. The order, unless modified by parentheses, is: NOT, AND, OR, XOR, IMP, and EQV. If more than one of the same operator exist in a given expression, they are evaluated from left to right. In other words two NOTs are performed first from left to right. This allows you to leave out many of the parentheses you've added to complex logical expressions. However, you may not want to remove all of the parentheses because they often help you understand the structure of the expression. You will find the following set of rules to be a good compromise between the two extremes.

1. NOT has the highest precedence of the logical operators. Therefore, you can omit the parentheses around NOT clauses and simple relational expressions that follow a NOT:

   NOT A=B AND NOT C=D

   rather than

   (NOT(A=B)) AND (NOT(C=D)).

   Remember to always put parentheses around complex expressions if the NOT applies to the whole expression.

2. You don't need to include parentheses around strings of simple relational expressions that are separated by a series of ANDs or ORs:

   A=B AND C=D AND E=G

   rather than

   (A=B AND C=E) AND E=F.

3. You should always use parentheses to clarify expressions that consist of mixed ANDs and ORs.

   (A=B AND C=D) OR (E=F AND G=H)

   is much clearer than

   A=B AND C=D OR E=F AND G=H.

Truth values are interpreted as numbers when they are referenced in a program. The following discussion will show how numbers are interpreted when they are supplied as truth values. The statement:

```
100 IF P THEN 500
```

means if P is any number other than zero, the program will branch to line 500. This statement forces the numeric value P to be taken as a truth value, which speeds up program execution. In a statement such as this, the numeric variable P has only two values. It is either TRUE or FALSE. When this is the case, P is called a flag.

Flags

A *flag* is a variable that has been assigned a truth value. Flags primarily transmit information about the workings of the program from one place in the program to another. A flag "remembers" a certain condition or occurrence at some point in the program so it can be acted upon at a later point.

−1 Represents True

Your interpreter uses a −1 to represent the value TRUE and 0 to represent the value FALSE, while some BASIC interpreters use 1 to represent the TRUE value. This is important to remember particularly when we discuss the internal representation of numbers and bit manipulation.

How Logical Operators Work at Machine Level

To understand how logical operators really work, you must look at the "machine level" operation of the interpreter. All forms of computer "data", including numbers, are stored as bit patterns. Bit patterns are arranged in groups of eight, called *bytes*. A byte is equal to eight bits, and each bit is identified by its position from the right. The logical operators perform simple logical operations on these bit patterns.

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

Machine Level Representation

In this version of BASIC, integer numbers from −32768 to +32767 are represented by two bytes (16 bits) at the machine level. The positive numbers 0 to 32767 are based on the powers of 2, instead of the powers of 10 in the decimal number system. In the decimal number system, each position to the left of the decimal point represents values 10 times greater than those in the position to the right. Similarly, in base-two or binary notation, each position represents values twice as great as those in the position to the right.

For example, the number 10101100 means

$1*2^7+0*2^6+1*2^5+0*2^4+1*2^3+1*2^2+0*2^1+0*2^0$

or

1*128+1*32+1*8+1*4=172

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

Two's Complement

BASIC uses a system called "two's complement" notation to represent negative numbers. In this system 01111111 11111111 represents 32,767 and 11111111 11111111 represents −1, not −32,767. If you continue, 11111111 11111110 represents −2. Following is a table of expanded bit pattern equivalence.

EQUIVALENCE TABLE

Decimal Equivalent	Two Byte Internal Representation		Two Byte Internal Representation		Decimal Equivalent
−1	11111111	11111111	00000000	00000000	0
−2	11111111	11111110	00000000	00000001	1
−3	11111111	11111101	00000000	00000010	2
−4	11111111	11111100	00000000	00000011	3
−5	11111111	11111011	00000000	00000100	4
−6	11111111	11111010	00000000	00000101	5
−7	11111111	11111001	00000000	00000110	6
−8	11111111	11111000	00000000	00000111	7
−9	11111111	11110111	00000000	00001000	8
−10	11111111	11110110	00000000	00001001	9
−11	11111111	11110101	00000000	00001010	10
−12	11111111	11110100	00000000	00001011	11
−13	11111111	11110011	00000000	00001100	12
−14	11111111	11110010	00000000	00001101	13
−15	11111111	11110001	00000000	00001110	14
−16	11111111	11110000	00000000	00001111	15
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
−32,765	10000000	00000011	01111111	11111100	32,764
−32,766	10000000	00000010	01111111	11111101	32,765
−32,767	10000000	00000001	01111111	11111110	32,766
−32,768	10000000	00000000	01111111	11111111	32,767

Table 5.12
Bit Pattern Equivalence

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range −32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or −1, logical operators return 0 or −1. The given operation is performed on these integers in bitwise fashion; i.e., each bit of the result is determined by the corresponding bits in the two operands.

NOT

When the interpreter encounters an expression such as NOT 14, the computer performs logical negation on each bit of the two-byte internal representation of the number 14 according to the truth table for NOT shown on Page 5-33. The truth table is repeated here with 1 and 0 instead of T and F.

X	NOT X
1	0
0	1

The NOT operation simply reverses the truth value of any given bit. Thus, 1 becomes 0 and vice versa. Therefore, 14 becomes NOT 14 as follows:

Internal rep. of 14	00000000 00001110
Internal rep. of NOT 14	11111111 11110001

The bit pattern equivalence table shows that the second bit pattern will be interpreted as the number −15, which is what the interpreter will print if PRINT NOT 14 is entered.

AND and OR

The operators AND and OR work in a similar manner on pairs of operands, according to the truth tables shown on Page 5.33. The following truth tables for AND and OR are written with 1 and 0 representing T and F respectively.

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

The interpreter evaluates the expression 5 AND 6, for example, by lining up the bit representations of each number and then applying the AND table to each corresponding pair of bits:

Internal rep. of 5:	00000000 00000101
Internal rep. of 6:	00000000 00000110
Internal rep. of 5 AND 6:	00000000 00000100

The result is interpreted as the number 4. At your computer the preceding example will look like this:

```
PRINT 5 AND 6
 4
Ok
```

The OR operator works the same way with the OR truth table:

Internal rep. of 5:	00000000 00000101
Internal rep. of 6:	00000000 00000110
Internal rep. of 5 OR 6:	00000000 00000111

The result corresponds to the number 7, as shown in the bit pattern equivalence table on Page 5.40.

XOR

When the interpretor encounters the XOR (exclusive OR) operator, it performs an evaluation based on the XOR truth table, repeated here using 0 and 1 instead of T and F.

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

The logical statement X XOR Y would appear as:

11 XOR 3

Internal rep. of 11 is:	00000000 00001011	
Internal rep. of 3 is:	00000000 00000011	
Internal rep. of 11 XOR 3:	00000000 00001000	(8)

The result corresponds to the number 8, as shown in the bit pattern equivalence table on Page 5.40.

IMP When the interpreter encounters an IMP (Implied) operator, it essentially combines the operations used in a NOT and OR evaluation. The logical statement X IMP Y is the same as NOT X OR Y. The truth table for an IMP operator is repeated here using 1 and 0 instead of T and F.

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

6 IMP 7

where 6 is:	00000000	00000110	
where NOT 6 is:	11111111	11111001	
where 7 is:	00000000	00000111	
where NOT 6 is:	11111111	11111001	
ORed to 7:	00000000	00000111	
equals	11111111	11111111	(−1)

The result is −1 when the expression is 6 IMP 7.

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

EQV

Finally, when the interpreter encounters EQV, it essentially performs two implication operations. The logical statement X EQV Y is the same as (X => Y) AND (Y => X). Through the law of implication, we arrive at (NOT X OR Y) AND (NOT Y OR X).

6 EQV 7			
where 6 is	00000000	00000110	
where NOT 6 is	11111111	11111001	
where 7 is	00000000	00000111	
where NOT 6 is	11111111	11111001	
ORed to 7	00000000	00000111	
is	11111111	11111111	(−1)
and			
where 7 is	00000000	00000111	
where NOT 7 is	11111111	11111000	
where 6 is	00000000	00000110	
where NOT 7 is	11111111	11111000	
ORed to 6	00000000	00000110	
is	11111111	11111110	(−2)
and where			
NOT 6 OR 7	11111111	11111111	(−1)
is ANDed to			
NOT 7 OR 6	11111111	11111110	(−2)
equals	11111111	11111110	(−2)

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

ARITHMETIC AND STRING OPERATORS

Arithmetic Operators and Expressions

63 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16 (binary 10000)
15 AND 14=14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
4 OR 2=6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)
10 OR 10=10	10 = binary 1010, so 10 OR 10 = 10 (binary 1010)
−1 OR −2=−1	−1 = binary 1111111111111111 and −2 = binary 1111111111111110, so −1 OR −2 = −1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of −1.
NOT X=−(X+1)	The two's complement of any integer is the bit complement plus one.

ARITHMETIC AND STRING OPERATORS

Numeric Functional Operators

BRIEF

A function is a predefined process or subprogram that takes one or more quantities as input and returns a single related quantity as output.

An intrinsic function is one of the functions built into the BASIC interpreter.

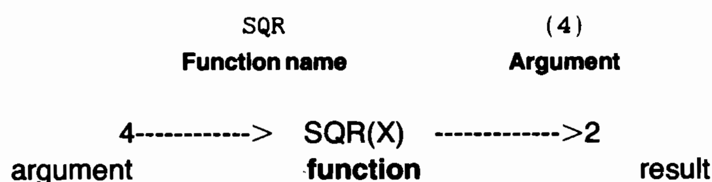
The interpreter calls the function and passes arguments to the function. The function processes the argument and returns the result.

Details

A function is used in an expression to call a predetermined operation that is to be performed on an operand. BASIC has intrinsic functions that reside in the system, such as SQR (square root) or SIN (sine). Following is an example of how functions work:

How Functions Work

You enter: **PRINT SQR (4)**
Computer Prints: 2



In the above description, the function performs a mathematical operation on the argument and returns the result. Table 5.13, on the next page, lists the numeric functions that are intrinsic to Z-BASIC.

ARITHMETIC AND STRING OPERATORS

Numeric Functional Operators

Intrinsic Numeric Functions

Standard Math Functions	**Result**
SQR(X)	Square Root of (X)
INT(X)	Nearest integer less than or equal to (X)
RND(X)	Randomize (X)
ABS(X)	Absolute value of (X)
SGN(X)	Sign of (X)
CDBL(X)	Convert (X) to a double precision number
CINT(X)	Convert (X) to an integer by rounding
CSNG(X)	Convert (X) to a single precision number
FIX(X)	Truncates decimal part of (X).
Exponentiation Functions	**Result**
EXP(X)	Raise e to the power of (X)
LOG(X)	Natural logarithm of (X)
Trigonometric Functions	**Result**
SIN(X)	Sine of angle (X), where (X) is in radians
COS(X)	Cosine of angle (X), where (X) is in radians
TAN(X)	Tangent of angle (X), where (X) is in radians
ATN(X)	Arctangent (in radians) of (X)

Table 5.13
Numeric Functions

ARITHMETIC AND STRING OPERATORS

Numeric Constants and Precisions

BRIEF

Constants are the actual values BASIC uses during execution.

Constants can be either numeric or string.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks.

Numeric constants can be stored internally as integers, single precision numbers, or double precision numbers.

Details

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

String Constants

A *string constant* is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Following are examples of string constants:

"HELLO"
"$25,000.00"
"Number of Employees"

ARITHMETIC AND STRING OPERATORS

Numeric Constants and Precisions

Numeric Constants

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants:

1. Integer constants — Whole numbers between −32768 and +32767. Integer constants do not have decimal points.

2. Fixed point constants — Positive or negative real numbers; i.e., numbers that contain decimal points.

3. Floating point constants — Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is 10^{-38} to 10^{+38}.

   Examples:

   235.988E−7 = .0000235988
   2359E6 = 2359000000

   (Double-precision floating point constants use the letter D instead of E.)

4. Hex constants — Hexadecimal numbers with the prefix & H.

   Examples:

   &H76
   &H32F

5. Octal constants — Octal numbers with the prefix &O or &.

   Examples:

   &0347
   &1234

ARITHMETIC AND STRING OPERATORS

Numeric Constants and Precisions

Single and Double-Precision

Fixed and Floating point constants may be either single-precision or double-precision numbers. Single-precision numeric constants are stored with six digits of precision, and printed with up to seven digits. With double-precision, the numbers are stored with 16 digits of precision and printed with up to 16 digits.

A single-precision constant is any numeric constant that has:

1. Seven or fewer digits; and/or
2. Exponential form using E; and/or
3. A trailing exclamation point (!).

A double-precision constant is any numeric constant that has:

1. Eight or more digits; and/or
2. Exponential form using D; and/or
3. A trailing number sign (#).

Examples:

Single-Precision Constants	Double-Precision Constants
46.8	345692811
−1.09E−06	−1.09432D−06
3489.0	3489.0#
225!	7654321.1234

For more information on integers, and single and double-precision values, see the following section on converting precisions. Also see "Variables," starting on Page 5.1.

ARITHMETIC AND STRING OPERATORS

Converting Numeric Precisions

BRIEF

Numeric constants may be either integers, single-precision, or double-precision numbers.

Single-precision numbers have up to seven digits.

Double-precision numbers can have up to 16 digits.

Each level of precision has a specific memory space requirement.

Details

Numeric constants may be integer, single-precision, or double-precision numbers. It is sometimes necessary to extend the precision of a number, according to what you want to do with that number.

Often it is necessary to change a double-precision number to a single-precision number or to change a single-precision number to an integer (whole number). Each level of precision requires less space than the level which precedes it. However, each level is less precise than the level that precedes it. It is important to remember to use consistent calculations within a program. It is often risky to mix precisions and maintain accuracy. You can go from double-precision to single-precision to integer without problems, but going from integer to single to double may yield an error.

Following is a list of the space requirements for each level of precision for variables, arrays, and strings.

Space Requirements

VARIABLES:	BYTES
INTEGER	2
SINGLE-PRECISION	4
DOUBLE-PRECISION	8

ARITHMETIC AND STRING OPERATORS

Converting Numeric Precisions

ARRAYS:	BYTES
INTEGER	2 per element
SINGLE-PRECISION	4 per element
DOUBLE-PRECISION	8 per element

STRINGS:

Three bytes overhead plus the present contents of the string.

From the space requirements listing you can see that a single-precision number takes up twice as much space as an integer does. And a double-precision number takes up twice as much space as a single-precision number. If your major concern is the conservation of space, you may use an integer. If your concern is with precise, accurate numbers, then you should use single or double-precision.

Type Declaration

Numeric variable names may declare integer, single, or double-precision values. The type declaration characters for these variable names are as follows:

- % Integer variable
- ! Single-precision variable
- # Double-precision variable

Conversion Rules

When you are converting a numeric constant from one type to another, keep the following rules and examples in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a `Type mismatch` error occurs.)

Example:

```
10 A% = 23.42
20 PRINT A%
RUN
 23
Ok
```

ARITHMETIC AND STRING OPERATORS

Converting Numeric Precisions

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

Examples:

```
10 D# = 6#/7
20 PRINT D#
RUN
 .8571428571428571
Ok
```

The arithmetic was performed in double-precision, and the result was returned in D# as a double-precision value.

```
10 D= 6#/7
20 PRINT D
RUN
 .8571429
Ok
```

The arithmetic was performed in double-precision, and the result returned to D (single-precision variable), was rounded and printed as a single-precision value.

3. Logical operators (see Page 5.32) convert their operands to integers and return an integer result. Operands must be in the range −32768 to 32767 or an `Overflow` error occurs.

4. When a floating point value is converted to an integer, the fractional portion is rounded.

Example:

```
10 C% = 55.88
20 PRINT C%
RUN
 56
Ok
```

5. If a double-precision variable is assigned a single-precision value, only the first seven rounded digits, of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single-precision value. The absolute value of the difference between the printed double-precision number and the original single-precision value will be less than 6.3E-8.

Example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
 2.039999961853027
 2.04
Ok
```

ARITHMETIC AND STRING OPERATORS

String Expressions and Operators

BRIEF

A string expression is composed of an operator, constants, variables, and functions.

A string constant (also known as string literal) is a sequence of characters up to 255 characters long, bounded by quotation marks.

String functions are commands that allow the manipulation of a string constant.

String variable names must have a dollar sign symbol ($) added to the end.

To concatenate two strings means to connect them into one string. The strings are connected by the concatenation operator, which is a plus sign (+).

Details

A *string expression* is formed like a numeric expression with the following components:

String operator
String constants
String variables
String functions

String Constants

A *string constant* is a sequence of up to 255 characters bounded by quotation marks.

Example:

You enter:	**PRINT "ABCDEFG123"**
Computer replies:	`ABCDEFG123`

Two strings may be joined together by the concatenation operator which is the plus (+) sign.

Example:

You enter:	**PRINT "ABCD" + "EFGH"**
Computer replies:	`ABCDEFGH`

ARITHMETIC AND STRING OPERATORS

String Expressions and Operators

String variable names are formed in exactly the same way that numeric variable names are formed, with the additional requirement that the string name must have the dollar sign symbol ($) added to the end.

You enter:	**A$ = "THIS IS A STRING"**
You enter:	**PRINT A$**
Computer replies:	THIS IS A STRING

Mixing Numeric And String Expressions

PRINT statements can contain numeric expressions or string expressions. You can also mix numeric and string expressions in a list, separating the expressions with commas or semicolons. Most versions of BASIC provide extra spaces when numeric expressions are separated by semicolons. Strings behave differently.

Example:

```
A = 1
B = 2
C = 3
A$ = "ONE"
B$ = "TWO"
C$ = "THREE"
PRINT A;B;C            RESULT:      1  2  3
PRINT A$;B$;C$                     ONETWOTHREE
PRINT A;A$;B;B$;C;C$                1 ONE  2 TWO  3 THREE
```

Both string and numeric expressions behave the same when separated by commas.

```
PRINT A,B,C,         RESULT:   1       2       3
PRINT A$,B$,C$                ONE     TWO     THREE
```

Notice that the leading blank usually seen in numeric values is reserved for a possible minus sign.

String Functions

String functions are used to manipulate a string constant. All string functions are referenced in detail in the reference guide of this manual. However, we will discuss a few of them here to give you an idea of how they work.

ARITHMETIC AND STRING OPERATORS

String Expressions and Operators

The MID$ creates a substring from a source string in the following manner:

You enter:	**A$ = "THIS IS A STRING"**
You enter:	**B$ = MID$ (A$,6,4)**
You enter:	**PRINT B$**
Computer replies:	IS A

Function name		**List of arguments**	
MID$	(A$,	6,	4)
	Source String	Starting Position	Number of Characters in substring

LEFT$ and RIGHT$ form substrings from the left end or right end of the source string. The starting point does not need to be specified for LEFT$ or RIGHT$ because it is implied by the length of the substring.

Example:

```
A$ = "ABCDEFG"
PRINT LEFT$(A$,2),RIGHT$(A$,2)    Computer prints: AB △△△△△△ FG
PRINT LEFT$(A$,4),RIGHT$(A$,4)    Computer prints: ABCD △△△△△ DEFG
```

NOTE: In this example the △ represents 2 spaces.

LEN is used to find the length of a string—that is, how many characters the string has.

VAL and STR$ are used to convert back and forth from a numeric value to the characters representing that value.

Example:

```
A$ = "2"
B$ = "3"
PRINT A$*B$
```

This creates an error condition called a "type mismatch" because A$ and B$ are string characters while the multiplication operator works only with numbers. However, you could use the VAL command to convert the string to a numeric value. In the following example, the "2" and "3" are converted to the numeric quantities 2 and 3 by the VAL function before multiplication is attempted.

```
PRINT VAL(A$)*VAL(B$)
```

ARITHMETIC AND STRING OPERATORS

String Expressions and Operators

The STR$ function goes the other way — it converts values numbers to their string representations.

To understand the last two string-related functions that we will discuss here, ASC and CHR$, you should recall that data is represented in the computer with bit patterns that form a binary code. (See "Logical Operators", starting on Page 5.32 for further information on bit patterns.) The system used to represent characters is called *ASCII* (American Standard Code for Information Interchange).

In BASIC, only the first 127 ASCII characters are used; therefore, each character is represented electronically by a unique seven-bit code. An ASCII conversion chart can be found in Appendix C of this manual.

Bit patterns can be interpreted in many different ways, depending on the code system you are using. You can interpret these patterns as characters, or as binary numbers or decimal numbers. The job of converting between these two interpretations is performed by the ASC and CHR$ functions.

The ASC function returns the decimal equivalent, while CHR$ function does exactly the opposite. Given a number within a certain range, it produces the corresponding character.

Again, this is just a summary of how string functions work. Detailed explanations for each string function can be found in the Alphabetical Reference Guide of this manual.

CHAPTER 6

FILE HANDLING

File Manipulation and Management

BRIEF

BASIC provides several sets of statements for creating and manipulating program and data files.

The file manipulation commands are very useful for manipulating program files. Some of these commands can also be used with data files.

The file management statements are used to open and close files, check for end-of-file, and to obtain information about the size of a file.

Details

FILE MANIPULATION COMMANDS

This is a review of the commands and statements that are useful for manipulating program and data files. These statements and commands are also discussed in the next two sections of this chapter.

```
FILES ["<filename>"]
```

FILES The FILES command lists the names of the files that are residing on the current disk. If the optional <filename> string is included, the names of the files on any specified disk can be listed.

```
KILL "filename"
```

KILL The KILL command deletes the file from the disk. "Filename" may be a program file, or a sequential or random access data file. If "filename" is a data file, it must be closed before it is killed.

```
LOAD "filename"[,R]
```

LOAD The LOAD command loads the program from disk into memory. The R option runs the program immediately. LOAD always deletes the current contents of memory and closes all files before loading. If R is included, however, open data files are kept open. Thus, programs can be chained or loaded in sections and can access the same data files.

FILE HANDLING

File Manipulation and Management

```
MERGE "filename"
```

The MERGE command loads the program from disk into memory but does not delete the current contents of memory. The filename must be saved in ASCII format. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory, and BASIC returns to Command Mode.

MERGE

```
NAME "oldfile" AS "newfile"
```

To change the name of a disk file, execute the NAME Command, NAME "oldfile" AS "newfile". NAME may be used with program files, random files, or sequential files.

NAME

```
RESET
```

RESET reads the directory information off of a newly inserted disk which you have exchanged for the disk in the current default drive. RESET does not close files that were opened on the former default disk. Therefore, use RESET only after you have closed any open files and replaced the current default disk.

RESET

```
RUN "filename"[,R]
```

RUN "filename" loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open.

RUN

```
SAVE "filename"[,A]
```

The SAVE command writes to disk the program that is currently residing in memory. The option writes the program in ASCII format. (Otherwise, BASIC uses a compressed binary format.)

SAVE

PROTECTED FILES

If you wish to save a program in an encoded binary format, use the protect option with the SAVE command. For example:

SAVE "MYPROG",P

Warning: A program saved this way cannot be listed or edited.

FILE MANAGEMENT STATEMENTS

BASIC provides a full set of I/O statements to be used for disk file management. These statements are listed in Table 6.1:

Statement	Function
OPEN	Opens a disk file and assigns a file number to the disk file.
CLOSE	Closes a disk file and de-assigns the file number from the disk file.
EOF	Returns - 1 (true) if the end of a file has been reached.
LOF	Returns the length of the file in bytes.
LOC	Returns the next record to be accessed for a random file and the total number of sectors or "records" accessed for a sequential file.

Table 6.1
File Management Statements

FILE HANDLING

File Manipulation and Management

The OPEN statement is used to assign a file number to a disk file name. The OPEN statement is also used to define the mode in which the file is to be used (sequential or random access).

The CLOSE statement performs the opposite function of the OPEN statement. It will de-assign the file number from a disk file name.

The EOF function will return −1 (true) if the end of a sequential file has been reached. The EOF function can also be used with random files to determine the last record number.

The LOF function returns the length of the file in bytes. LOF divided by the length of a record is equal to the number of records in the file.

The LOC function, when used with a random file, will return the next sector to be accessed. When it is used with a sequential file, it returns the number of records accessed since the file was opened.

These statements are discussed along with specific examples that utilize these statements in "Sequential Data Files" (Page 6.5) and "Random Access Files" (Page 6.16).

FILE HANDLING

Sequential Data Files

BRIEF

A sequential data file is a file that must be accessed in a sequential order, starting at the beginning of the data block and proceeding in order until an end-of-data marker is encountered or the required number of items has been read.

Sequential files are easier to create than random files, yet they are limited in terms of speed and flexibility.

The BASIC interpreter communicates with I/O buffers, which are reserved spaces in memory, maintained by the operating system for holding incoming or outgoing data.

The data found in a sequential file can be retrieved, formatted, updated and manipulated in a variety of ways.

Details

Sequential files must be accessed in the same sequential order that they were written, starting at the beginning of the data block and proceeding in order until an end-of-data marker is encountered or the required number of items have been read.

Since the items must be read in order, this kind of data organization is called *sequential*. It works fine for applications that process a batch of data in a certain order, but not so well when you need to access individual items within a data block. For this, you need a random access file structure (see Page 6.16).

Just as DATA statements are a simpler and more fundamental form of data organization than arrays, sequential files are a simpler and more fundamental form of storage than random-access files.

FILE HANDLING

Sequential Data Files

Buffers

BASIC never "sees" the file of the disk unit. In fact, BASIC never sees any of the I/O devices attached to the computer. Instead, BASIC sees a buffer, a reserved space maintained by the operating system for holding incoming or outgoing data. One buffer might hold data coming in from the keyboard, another might hold data coming from or going to a particular disk file, and go on.

It doesn't make any difference to the BASIC interpreter how input data gets into an I/O buffer or what happens to output data once it's placed there; all the interpreter needs to know about the devices is where the corresponding buffers are located in memory. Everything else is the responsibility of the operating system and its various device drivers.

Physical Records

The fixed amount that can be physically written to or read from the disk at one time is 256 bytes. This fixed amount of data is called a *physical record.* The physical record is the same length as a disk sector, 256 bytes, so you can think of a file buffer full of data as "one sector's worth."

There are two types of file buffers used with sequential files: input buffers and output buffers. The buffers themselves are identical, but BASIC must be told whether a given buffer is to be used for input or output.

Output Buffers

When BASIC appears to be writing a sequence of data to a disk file, it's really placing one data item after another in an output buffer. When the output buffer is full, Z-DOS writes the entire physical record to the disk, resets a pointer to the beginning of the buffer, and waits until the buffer fills up again before it writes another physical record.

Input Buffers

Input works in a very similar way. When BASIC appears to get data from a sequential disk file, Z-DOS is really reading one physical record at a time from the disk, and placing it in a way that is similar to the way it would read a DATA statement or a line from the keyboard. When the contents of the buffer have been exhausted, Z-DOS reads another physical record from the disk, places it in the input buffer, and so on until it reaches the end of the file or BASIC stops requesting data items from the buffer.

CREATING A SEQUENTIAL DATA FILE

The statements and functions that are used with sequential files are:

OPEN	PRINT#	INPUT#	WRITE#
	PRINT# USING	LINE INPUT#	
CLOSE	EOF	LOC	

Table 6.2
Sequential File Statements and Functions

OPEN Statement

To create a sequential disk file (or read from one), you must designate a disk I/O buffer with the OPEN statement. The OPEN statement requires three items of information: the mode (Input or Output); the number of the disk I/O buffer; and the file specification, which tells BASIC where to start accessing the disk, according to the sector pointers maintained in the disk directory.

You must give the extension if there is one, and if necessary, you must also give the drive number to distinguish between two files that have the same name. The I/O buffer is specified as 1, 2, or 3. One of the three file buffers automatically created by the DOS is associated with the specified physical disk file. The Input or Output mode is also given by one of the designators "I" or "O". Thus, the statement:

```
OPEN "O",1,"TEST.ASC"
```

will designate buffer 1 as an output buffer and tell Z-DOS to associate this buffer with the physical file it knows as "TEST.ASC". The ASC extension represents a naming convention for a mixed ASCII data file that contains both string and numeric data. You can use any extension that you like.

EOF Pointer

In addition to setting up a buffer, the OPEN statement causes Z-DOS to reset an essential directory pointer which points to the last physical record in the file. This pointer, called EOF (end-of-file), is now set to point to the "zero" record, which is the very beginning of the first record. This indicates that the file contains no physical records (written disk sectors) yet. Since the beginning of the file TEST.ASC is now the same as the end of the file as far as Z-DOS is concerned, any preexisting file by that name disappears.

FILE HANDLING

Sequential Data Files

Procedure

The program steps listed in Table 6.3 are required to create a sequential file and access the data in the file.

Step	Program statement
1. OPEN the file in "O" mode.	`OPEN "O",#1,"TEST.ASC"`
2. Write data to the file using the PRINT# statement. (WRITE# may be used instead.)	`PRINT#1,A$,B$,C$`
3. To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode.	`CLOSE #1` `OPEN "I",#1,"TEST.ASC"`
4. Use the INPUT# statement to read data from the sequential file into the program.	`INPUT#1,X$,Y$,Z$`

Table 6.3

Creating a Sequential File—Program Steps

PRINT #

Since we have already discussed the OPEN statement, we will now discuss the second step, the PRINT# statement, which is used to write the data to the file. If, for example, you have just OPENed a file, you would want BASIC to supply a string of characters to the output buffer, just as if a string of characters were being sent out to be printed on the display. You would use an expanded version of the PRINT statement called PRINT#. This works like the usual PRINT statement, except that you must include the buffer number immediately following the PRINT keyword.

Suppose, for example, you wanted to write the following set of data items to the disk:

A=1.1 : B=2.22 : C=3.333 : D=4.444 : E=5.5555

NOTE: It would probably be most helpful if you try this yourself by opening a file named TEST.ASC in the immediate mode with the OPEN statement repeated below.

```
OPEN "O",1,"TEST.ASC"
```

Nothing happens when you write the five data items A,B,C,D,E to the disk, because you haven't filled the 256-character output buffer. Use the up arrow to position the cursor under the P in PRINT and then press **RETURN** to repeat the PRINT# statement, which will enter a few more sets of data into the buffer.

```
PRINT #1,A,B,C,D,E
Ok
```

CLOSE

Eventually, you will hear the disk drive click when you fill the buffer and the record gets written. If you entered a few more of these PRINT# statements, creating a partially filled buffer, you have to finish the process by entering the word **CLOSE**.

This writes the second physical record to the disk, updates the EOF pointer, and releases buffer #1 for something else. CLOSE by itself closes all open disk files. If you had more than one file open and didn't want to disturb the others, you would enter **CLOSE 1**. You should make a habit of always closing files; otherwise, an OPEN statement could result in a `File Already Open` error message.

The key to understanding the PRINT# statement is the fact that it sends the same set of characters to the disk that the corresponding PRINT statement would send to the terminal. To see what each of the statements you entered has written to the disk, enter the corresponding PRINT statement in immediate mode and look at the output.

```
PRINT A,B,C,D,E
 1.1          2.22          3.333          4.444          5.55555
Ok
```

The series of characters you see on the display, including the string of spaces between each pair of numbers, is exactly what each PRINT# statement puts on the disk.

FILE HANDLING

Sequential Data Files

INPUT #

Characters are received from the disk the same way that they are received from the terminal, except you use INPUT# instead of INPUT. To read data from the file you created in the preceding example, open the file for input with the following statement:

```
OPEN "I",1,"TEST.ASC"
```

This designates I/O buffer 1 as the sequential input buffer for the file "TEST.ASC" and resets a Z-DOS pointer to the beginning of the disk file. This operation is the disk equivalent of a RESTORE to the beginning of a block of DATA statements.

Now you can read a series of data items from the disk file and assign them to a series of variables in the same way they would be read from keyboard input.

The following short program creates a sequential file, "DATA", from information you input at the terminal:

Sample Program 1

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN
NAME? MIKE JONES
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? MARY SMITH
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? ALICE ROGERS
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? ROBERT BROWN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? DONE
Ok
```

Program 1

Create a Sequential Data File

Now look at Program 2. It accesses the file "DATA" that was created in Program 1 and displays the name of everyone hired in 1978:

Sample Program 2

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
ALICE ROGERS
ROBERT BROWN
Input past end in 20
Ok
```

Program 2

Accessing a Sequential File

Program 2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an `input past end` error messages. To avoid getting this error message, insert line 15, which uses the EOF function to test for end-of-file:

```
15 IF EOF (1) THEN END
```

and change line 40 to **GOTO 15**.

The word *terminator* refers to the same thing as the word *delimiter*. It is a special character that marks the boundary of a data item, like the commas used to separate items in DATA statements.

Terminator

The word terminator is used in this discussion for two reasons. First, it throws the emphasis on the function of marking the end of an item of data rather than that of separating two adjacent items and second, it includes certain conditions as terminators as well as the special characters usually referred to as delimiters. For your purposes, a *terminator* is any condition, character, or set of characters that will make INPUT# conclude that it has reached the end of the series of characters that represent a given item of data in a disk file.

FILE HANDLING

Sequential Data Files

There are only two terminators that will always cause INPUT# to stop accepting characters as part of a given item of data, and both of these universal terminators are conditions rather than characters. They are:

1. The last character in a file. INPUT# will not attempt to read the last item past the end of a file.

2. The 255th character in an item of data. INPUT# will not attempt to read more characters than will fit in a single string.

Essentially, there are three different forms that data can take when stored in a BASIC sequential file. They are: numeric data, strings that are not enclosed in quotation marks, and strings that are enclosed in quotation marks.

The usual item terminator in all three cases is the comma. With INPUT#, however, the set of acceptable item terminators is somewhat different for each storage type. For numeric data, the usual item terminator is a space, or set of spaces. For unquoted strings, the usual item terminator is the comma; and for quoted strings the quotation mark at the end of each string is the usual terminator.

The differences between the terminators mean that slightly different techniques will have to be used to form the PRINT# statements used for each type.

Numeric Data

You will recall from our TEST.ASC example that numeric data items are stored with one or more spaces. The statement, PRINT #A,B,C,D,E was stored as follows:

```
PRINT A,B,C,D,E
 1.1          2.22         3.333        4.444          5.55555
Ok
```

This form is an unnecessary waste of room on the disk, because INPUT# will accept as little as one space as a valid numeric terminator. Consequently, it is better to use the semicolon terminator (PRINT #1,A;B;C;D;E) to put just one or two spaces between items. Semicolons between the variables in the PRINT # statement produce a series of characters that is identical to the earlier version as far as INPUT # is concerned, but takes up less space on the disk.

A numeric item input will also terminate if a RETURN is encountered.

Unquoted Strings

When you are using PRINT # and INPUT # with unquoted strings, keep in mind that spaces do not terminate a string read by INPUT# , which means you can include spaces as part of the string itself (if placed after the first significant character). The character you should use to properly end each string is the comma.

The basic method for using commas is to insert a comma (",") with quotes into the PRINT # list wherever a comma without quotes should appear in the disk image. The INPUT# will then read back each string as terminated by its comma. Just as with numeric data, the form PRINT #1,A$,",",B,",",C$ should not be used. You should substitute semicolons for commas as variable list delimiters so unwanted strings of spaces won't be created in the disk image.

Another terminator that works with unquoted string data is RETURN. You don't need a comma to terminate the last item in the PRINT # statement. The RETURN added to the end by PRINT # will automatically terminate an unquoted string like C$. This properly ends input of the string when it's read back later and separates it from whatever might follow it in the file.

Quoted Strings

String expressions are enclosed with quotation marks (") to avoid confusion when other terminators such as commas are used within the string you are trying to input as data. When INPUT # encounters a quotation mark as the first significant character in a string item, it takes this as a direction to include all the following characters up to the next quotation mark as part of the string. This allows you to put commas, RETURNS, or any other character you like into the data string. The single exception is the quotation mark, that ends it.

FILE HANDLING

Sequential Data Files

Formatted Data

A program that creates a sequential file can also write formatted data to the disk with the PRINT # USING statement. The PRINT # USING statement is fully documented in the reference guide. It is mentioned here to advise you of the capability of formatting the data in your sequential files to the format that you specify. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

ADDING DATA TO A SEQUENTIAL DATA FILE

Updating

If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents. You can use the following procedure to add data to an existing file called "NAMES":

1. OPEN "NAMES" in "I" mode.
2. OPEN a second file called "COPY" in "O" mode.
3. Read in the data in "NAMES" and write it to "COPY".
4. CLOSE "NAMES" and KILL it.
5. Write the new information to "COPY".
6. Rename "COPY" as "NAMES" and CLOSE .
7. Now there is a file on disk called "NAMES" that includes all the previous data plus the new data you just added.

Program 3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustrates the use of LINE INPUT# to read strings with embedded commas from the disk file. Remember, LINE INPUT# will read in characters from the disk until it sees a carriage return (it does not stop at quotes or commas) or until it has read 255 characters.

Sample Program 3

```
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$="" THEN 200 'CARRIAGE RETURN EXITS INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
215 END
2000 IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"COPY":RESUME 120
2010 ON ERROR GOTO 0
```

Program 3

Adding Data to a Sequential File

The error trapping routine in line 2000 traps a `File not found` error message in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

FILE HANDLING

Random Access Files

BRIEF

Random-access files are accessed randomly, which makes it unnecessary to read through all of the data records to get to a specific data record.

Although creating a random-access file involves more program steps than a sequential file, the speed, flexibility, and the efficient use of storage space are distinct advantages.

The fundamental storage unit is called a record. Records are usually numbered to permit random access.

Random-access storage and retrieval takes place through a buffer.

Details

The biggest advantage to random files is that data can be accessed randomly; i.e., anywhere on the disk — it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records, and each record is numbered.

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. Random files require less room on the disk because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

Random Files Are Like Arrays

A random-access file is very much like an array: both consist of a collection of numbered units, any one of which you can immediately access simply by specifying its number. In the case of random-access files, the numbered units are much more complex than in the case of arrays. The fundamental storage unit in an array is the individual array element, which is a single item of data. The corresponding unit in a random-access file is the record, which can have several items of data.

Random Access Files

Buffers

Like all forms of disk I/O, random-access storage and retrieval takes place through a buffer. Just as in the case of sequential I/O, the buffer used in random-access I/O is a fixed-length section of memory that holds data coming from or going to the disk in a form that can be handled by the interpreter. There are, however, some important differences between random-access and sequential buffers in the way they are used.

First, once you assign a random-access buffer to a file by an OPEN statement, you can use it for both input and output. You can use sequential buffers for either input or output, but not both.

Second, random-access buffers are not written to or read from the disk automatically as sequential buffers are. In random-access files, the buffer and the disk are accessed by two separate processes. You must explicitly specify the operations of reading a record into the buffer or writing a record from the buffer to the disk by means of the GET and PUT keywords.

Finally, the buffer is organized differently in these two forms of disk access. In a sequential buffer, the arrangement of data is not fixed, but is specified by delimiters or terminators. That is, sequential buffers are delimiter-structured. By contrast, random-access buffers are field-structured. Each data item occupies a predefined section of the buffer called a field. Also, external pointers access these buffer fields rather than internal delimiters.

Statements and Functions

The statements and functions that are used with random files are:

OPEN	FIELD	LSET/RSET		GET
PUT	CLOSE	LOC	LOF	EOF
MKI$	CVI			
MKS$	CVS			
MKD$	CVD			

Table 6.4

Random File Statements and Functions

FILE HANDLING

Random Access Files

CREATING A RANDOM FILE

Procedure

The program steps in Table 6.5 are required to create a random file.

1.	OPEN the file for random-access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.	`OPEN "R",#1,"FILE",32`
2.	Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.	`FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$`

3.	Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single-precision value, and MKD$ for a double-precision value.	`LSET N$=X$`
		`LSET A$=MKS$(AMT)`
		`LSET P$=TEL$`

4.	Write the data from the buffer to the disk using the PUT statement.	`PUT #1,CODE%`

Table 6.5

Program Steps for Creating a Random File

Now that you know the statements and functions used in random-access files an the order in which they are used, we'll discuss opening a file for random-access in detail.

OPENING A FILE FOR RANDOM-ACCESS

As in the case of sequential files, you must associate a particular buffer with a particular disk file and specify the buffers for both input and output. You indicate their mode of operation by the single specifier "R". For example, to open a disk file named "INVNTRY.DAT" for random-access and associate it with buffer number 1, you would enter:

```
OPEN "R",#1,"INVNTRY.DAT"
```

OPEN Statement

Unlike the sequential OPEN "O" statement, this will not automatically kill a previously existing file with that name. If no such file exists, OPEN "R" will automatically create one. You cannot use random-access techniques on a sequential file and vice versa.

In addition, a parameter at the end of the OPEN statement specifies the size of the buffer in bytes.

STRUCTURING THE RANDOM BUFFER INTO FIELDS

FIELD Statement

The record contained in the random-access buffer must be subdivided into fixed length-fields. Random records are like string records in the sense that they can be accessed only through string variables. The FIELD statement divides the characters in the buffer into a certain number of fields, each consisting of a specified number of characters and referenced by a string variable. The statement has the general form:

```
FIELD BU%, N1% AS A1$, N2% AS A2$,...
```

Where BU% stands for the number of the random-access buffer, N1% for the number of characters in the first field, A1$ for the string, N2% for the number of characters in the second field, and so on. Thus, you could implement the field structure for an inventory program as follows:

```
FIELD#1, 1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
```

FIELD#1

Divides the record read from file #1 into the five sections that follow:

1 AS F$
The first character can be referenced as F$.

30 AS D$
The next 30 characters can be referenced as D$.

2 AS Q$
The next 2 characters can be referenced as Q$.

2 AS R$
The next 2 characters can be referenced as R$.

4 AS P$
The last 4 characters can be referenced as P$.

This statement divides the first 39 characters of buffer #1 into five fields, which can each be referenced separately using their variable names. This only affects the first 39 characters in the buffer. The rest are left undefined and are wasted if the record size is more than 39 characters.

ASSIGNING DATA TO FIELDS AND WRITING THE BUFFER TO THE DISK

The FIELD statement sets up a system of pointers into a series of character locations that permit you to refer to the contents of each field by name. Therefore, PRINT P$ will print the last four characters in the buffer which have been assigned to the P$ variable.

For example, in an ordinary string, the statement PRINT A$ instructs BASIC to consult an internal table of string pointers to reference the particular section of string space associated with the name A$. The difference between referencing an ordinary string and referencing a FIELD string is that the latter has a fixed length and that its pointer indicates the section of memory reserved for buffers rather than the section reserved for strings.

It is for this reason that you cannot use the LET, INPUT, or READ statements to assign values to field strings, because these statements will not put the characters into the buffer. To properly store the strings in the buffer, you must use one of the special buffer assignment keywords LSET or RSET.

LSET Statement

The LSET statement instructs BASIC to store the given characters in the buffer field specified by the given field name, starting at the leftmost end of the field. For example,

```
LSET NA$="N BENCHLEY"
```

where NA$ is a name string that has been assigned 16 character positions and will create the series of characters:

```
N^BENCHLEY^^^^^^
```

in the first 16 character positions of the buffer. The symbol ^ denotes a space character. You don't need to include the six trailing spaces used to "pad out" the 16 character name. They are automatically supplied by LSET.

FILE HANDLING

Random Access Files

Notice also that LSET begins to assign characters at the first (leftmost) position in the field. This is called "left-justified" within the NA$ field. If the string assigned to NA$ is shorter than the length of the field, as in this example, LSET adds spaces on the right. If the string is longer, LSET will chop off or truncate the excess right-hand characters.

RSET Statement

RSET works like LSET except that the string is right-justified in the name field, if the string is shorter than the field length. Thus, the statement:

```
RSET NA$="N BENCHLEY"
```

creates the series of characters:

```
^^^^^^N^BENCHLEY
```

However, RSET will not truncate the excess characters on the left. Instead, it will truncate the excess characters on the right, just as LSET will.

PUT Statement

To write a record from a random buffer to a random file, you must use the PUT statement. A sequence of immediate mode statements is shown below that will open a random-access file, set up a field structure for your address records, and place one of these records in the buffer. After which you will see why the PUT statement is necessary.

```
F$="ADDRESS.DAT"
Ok

OPEN "R",#1,F$
Ok

FIELD #1, 16 AS NA$, 33 AS SA$, 14 AS CY$, 8 AS SZ$
Ok

LSET NA$="N BENCHLEY"
Ok

LSET SA$="12 ASHMONT AVE APT 6"
Ok

LSET CY$="NEWTON"
Ok

LSET SZ$ "MA 02158"
Ok
```

If you continued to place records in the buffer using a series of LSET statements, you would overwrite the data in the buffer with different data. In other words, records are not automatically written to the disk, as in the case of sequential files. To write the contents of buffer #1 to the disk, you must enter the statement:

```
PUT #1
Ok
```

Now you can add more data to your "ADDRESS DAT" file without overwriting the information already in the buffer.

```
LSET NA$="A DUFFY"
Ok

LSET SA$="233 AUSTIN DR."
Ok

LSET CY$="OAK PARK"
Ok

LSET SZ$="IL 66699"
Ok

PUT #1
Ok

LSET NA$="J POPE"
Ok

LSET SA$="3100 BROADWAY"
Ok

LSET CY$="NEW TOWN"
Ok

LSET SZ$="IL 60657"
Ok

PUT #1
Ok

CLOSE
Ok
```

At this point, you have written three address records to the file and closed it. Next, you will retrieve the three records.

GETTING RECORDS OUT OF THE FILE

GET Statement

To retrieve records from the file that you have stored, you must perform the following steps:

1. Open the file for random access (if it is not already open) and set up field variables with an appropriate FIELD statement.

2. Read each record into the buffer with the keyword GET.

3. Process data in given fields of the record by referencing the corresponding field variables.

Using the preceding example, you could retrieve the records from the file you created by opening the file as follows:

```
OPEN "R",#1,F$
Ok

FIELD #1, 16 AS NA$, 33 AS SA$, 14 AS CY$, 8 AS SZ$
Ok

GET #1
Ok

?NA$;SA$;CY$;SZ$
N BENCHLEY  12 ASHMONT AVE APT 6  NEWTON  MA 02158
Ok

GET #1
Ok

?NA$;SA$;CY$;SZ$
A DUFFY        233 AUSTIN DR.        OAK PARK  IL 66699
Ok

GET #1
Ok

?NA$;SA$;CY$;SZ$
J POPE          3100 BROADWAY        NEW TOWN  IL 60657
Ok

CLOSE
Ok
```

FILE HANDLING

Random Access Files

This example shows that, as each record is brought into buffer #1 by the GET 1 statement, the fields of that record are automatically assigned to the field variables NA$ and so on simply because pointers into the buffer have already been set up by the FIELD statement. The practical effect of this is that the data in each field are immediately accessable through the corresponding variable as soon as the record is read into the buffer, without needing a separate statement like INPUT # to connect a given item (field) to a variable name.

Otherwise, this example doesn't seem to differ that much from a series of reads done on a sequential file. You put in three records in order and got three records back out again in the same order. This apparent similarity comes about only because we chose to default to a sequential kind of access by using incomplete forms of the PUT and GET statements.

More About PUT

The complete form of the PUT statement is PUT BU%,REC%, where BU% is a buffer number and REC% is the number of a given record. The statement PUT 1,23, for instance, means write the current contents of buffer #1 to disk as record 23. If you omit the specified record, as in the example you saw before, the interpreter automatically assumes a record number one greater than that of the "current record," which is the last record written or read from the disk.

When you open the file, the default "current record" is zero. A following PUT without a specific record given will access the "next" record number 1. Therefore, the three PUT statements in this example - PUT1... PUT1...PUT1... are by default equivalent to PUT 1,1...PUT 1,2...PUT 1,3 and have therefore stored the three test records as records 1, 2, and 3 in the file.

More About GET

The complete GET statement has the very similar form GET BU%,REC%, where BU% and REC% stand for the buffer number and record number, respectively. Just as with the PUT statement, the interpreter will assume a record number one higher than the last record accessed by GET or PUT if you leave out the explicit record number of the GET statement. Since you began the last example by reopening the file, the "current record" at the beginning defaults to, and the series of statements GET1...GET1...GET1 was equivalent to the three statements GET 1,1....GET 1,2...GET 1,3.

LOF Function

It is often useful to know the last record number in a random-access file. This number is returned by the LOF, or "last-of-file" function.

LOC Function

The function call LOC(2), for instance, will return the record number of the last numbered record in the file associated with buffer #2. You can use this information to terminate a read of the records in the file or to tell you where to begin adding new records.

STORAGE AND RETRIEVAL OF NUMERIC DATA

Converting Numeric Quantities

Since all random-access storage and retrieval is done through string variables, you cannot store numeric quantities directly in random-access disk files. They must somehow be converted to string representations before they can be placed in a field and put on a disk. One way you can do this is to convert the internal binary representation of the number to a string (ASCII) representation using the STR$ function before placing it in the buffer.

You would then use the VAL function to convert from the series of characters back to a binary-encoded numeric quantity when reading the number back from the disk. However, using this method would be both inefficient and wasteful; inefficient because it takes time for the interpreter to translate the series of ASCII characters to binary (and vice versa), and wasteful because of the fixed-length field in which the ASCII representation must be stored, regardless of the varying number of ASCII characters into which the numeric quantity would actually be translated.

Instead of STR$ and VAL, BASIC provides a special set of functions that allow the bytes that make up a binary number to be directly assigned to a string variable as if they were characters and, conversely, allows the characters stored in a numeric data field on the disk to be directly read back to memory as the bytes that make up the internal representation of a number.

Make String Functions

This change is performed by the three "make compressed string" functions MKI$, MKS$, and MKD$. MKI$ converts a two-byte integer to a two-byte string. MKS$ converts a four-byte single-precision number to a four-byte string, and MKD$ converts an eight-byte double-precision number to an eight-byte string.

Conversion Functions

When you are reading numbers back from a random-access file, you must change them from strings back to numbers before they can be assigned to numeric variables. This is accomplished by three conversion functions, CVI, CVS, and CVD. CVI changes a two-byte string into an integer, CVS changes a four-byte string into a single-precision number, and CVD changes an eight-byte string into a double-precision number.

Application

The inventory program starting on the next page illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

FILE HANDLING

Random Access Files

```
120 OPEN"R",#1,"INVEN.DAT",39
125 FIELD#1,1AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1) OR (FUNCTION>6) THEN PRINT
      "BAD FUNCTION NUMBER":GOTO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$:
    IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
```

```
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;" IN STOCK": GOTO 600
630 Q%=Q%-S%
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;
     " REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)< CVI(R$) THEN PRINT D$;" QUANTITY";
      CVI(Q$) TAB(50); "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF (PART%<1)OR(PART%>100) THEN PRINT "BAD PART NUMBER":
    GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

Sample Inventory Program

CHAPTER 7 PLOTTING COORDINATES

The Video Screen

BRIEF

The screen display format has 25 lines numbered 1–25, and a width of 80 columns numbered 1–80.

The video resolution of the Z-100 is 640 horizontal addressable points, and 225 vertical addressable points.

Vertical points on the screen are associated with the Y axis, and horizontal points are associated with the X axis.

The screen can be changed to H-19 graphics mode or reverse video, via the SCREEN statement. (See Page 7.3).

Format: `Screen [graphics,] [reverse video]`

The SCREEN function returns the ASCII value of a character on the screen at the specified location. (See Page 7.5).

Format: `X = SCREEN(row,col [,z])`

Details

The first step in using Z-BASIC graphic capabilities is to understand the characteristics of the video screen and how to plot coordinates on it.

The Z-100 All-in-One Monitor has a 12-inch diagonal screen. The display format has 25 lines numbered 1–25, and a width of 80 columns numbered 1–80.

Video resolution is the density of the individual pixels (points) on the screen. The video resolution of the Z-100 is 640 horizontal addressable points, and 225 vertical addressable points. This high resolution permits sharper and more detailed graphic images to be displayed on the screen.

PLOTTING COORDINATES

The Video Screen

Vertical points on the screen are associated with the Y axis. Horizontal points are associated with the X axis. To plot or locate coordinates on the screen you should first understand the orientation of the coordinates on the X and Y axis. Point 0,0 is the first point in the top left corner of the screen. Point 639,0 is the top right point. Point 0,224 is the bottom left point and 639,224 is the bottom left point as illustrated in Figure 7.1.

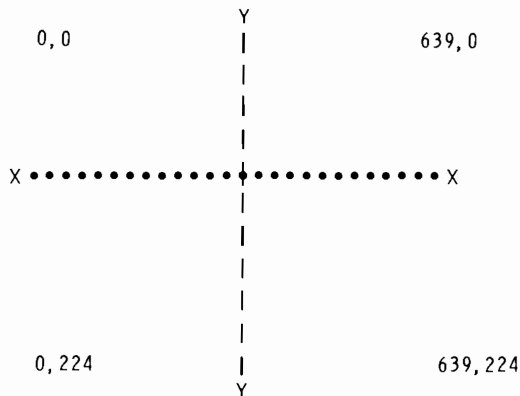


FIGURE 7.1
X,Y Coordinates of the Four-Corner Points

We will discuss plotting coordinates throughout this chapter, since many Z-BASIC statements use the X,Y coordinates as arguments. In particular, in our discussion of the LINE statement (Chapter 8), we will show you how to use these four points to draw a border on your screen.

SCREEN STATEMENT

The SCREEN statement allows you to put Heath/Zenith H-19 graphic characters on the video display and also permits the use of reverse video. H-19 graphics have been included to assure compatibility with software programs using H-19 graphics.

Reverse video will print black characters on a white background. This is a convienent feature for highlighting text and other special affects.

Format: `Screen [graphics,] [reverse video]`

Graphics when it appears in the SCREEN statement 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 be omitted. Omitted parameters assume the previous value.

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           'reverse video off.
```

PLOTTING COORDINATES

The Video Screen

If you are using H-19 graphics for the first time, we advise that you draw your graphic image first on a video layout grid. (You can make one by drawing a grid with 25 vertical boxes by 80 horizontal boxes.) After you have the graphics on paper it will be easier to transfer them to the screen.

In H-19 graphic mode, lower case letters are converted to graphic symbols. Therefore you must refer to the Graphics Symbol Table in Appendix C of this manual. After you decide which graphic character you want to use, note it's lower case letter equivalent, and input this letter to BASIC. BASIC will then convert this letter to the corresponding graphic symbol.

Example:

```
10 CLS
20 SCREEN 1
30 PRINT "faac"
40 PRINT "eaad"
50 SCREEN 0
```

When this program is run, faac will convert to the top half of a small box. In line 40, eaad will convert to the bottom half of a small box.

If H-19 graphics are in effect while in direct mode, all lowercase alphabetic characters typed will produce H-19 graphic characters. Therefore, programs using H-19 graphics should clear graphic mode before returning to command mode, as done in line 50 of the example above.

You can use the locate statement described on Page 7.12 to place the box in the center of the screen by changing lines 30 and 40 to:

30 LOCATE 12,38:PRINT "faac"
40 LOCATE 13,38:PRINT "eaad"

SCREEN FUNCTION

The SCREEN function returns the ASCII value of the character that is located at the specified row and column on the screen.

Format: `X = SCREEN(row,col [,z])`

X is a numeric variable receiving the integer returned.

row is a number between 1 and 25, the row number.

col is a number between 1 and 80, the column number.

z is an optional number between 0 and 255, which, if present and not zero, will cause the function to return the color attributes of the location instead of the ASCII value of the character.

NOTE: Any values entered outside these ranges will result in an `Illegal Function Call` error.

Action:

The integer value of the ASCII character at the specified location is stored in the variable. If the optional parameter <z> is given and not zero, a single byte, containing color attribute information is returned.

Example:

`10 CLS`	Clear the screen.
`20 COLOR 4,7`	Set foreground attribute to red (4) and background attribute to white (7).
`30 PRINT "H"`	Print the ASCII character H in the top left corner of the screen (location 1,1).
`40 X=SCREEN(1,1,1)`	Use the screen function to determine the color attributes of H (at location 1,1).
`50 PRINT HEX$(X)`	Print the hexadecimal value of the result.
`60 Y=SCREEN(1,1)`	Use the screen function to return the ASCII value of the character at 1,1.
`70 PRINT Y`	Print the result.

```
RUN
74
 72
Ok
```

PLOTTING COORDINATES

The Video Screen

In this example, an H is printed in the top left corner of the screen. The screen function is then used to determine the color attributes of H. To interpret the results, you must use the HEX$ function to convert the decimal variable stored in X to the hexadecimal equivalent. The first digit (a 7 in this case) tells you the background color attribute (white), while the second digit (a 4) indicates the foreground color attribute (red).

The second number that is printed is the ASCII value of H (79). To verify that this is correct, you could either look up the character in the ASCII table in Appendix C or you could use the CHR$ function in the following manner:

PRINT CHR$ (72)

```
H
Ok
```

PLOTTING COORDINATES

Locating and Activating Pixels

BRIEF

Three statements in Z-BASIC that use the X and Y pixel coordinates as arguments are POINT, PSET, and PRESET.

The POINT function allows you to read the attribute value of a pixel from the screen.

Format: `POINT (X,Y)`

The PSET statement is used to turn on a point at a specified location on the screen.

Format 1: `PSET (X coordinate , Y coordinate) [,attribute ]`
Format 2: `PSET STEP (X offset, Y offset)`

The PRESET statement is used to turn off a point on the screen at a specified location.

Format 1: `PRESET (X coordinate, Y coordinate) [,attribute]`
Format 2: `PRESET STEP (X offset, Y offset)`

Details

Now that you are familar with the orientation of the coordinates and characteristics of the screen you will be able to locate pixels and turn them on or off.

The POINT function allows the user to read the color value of a pixel from the screen. The format of the POINT function is:

```
POINT (X,Y)
```

If the point given is out of range the value `-1` is returned. Valid returns are any integer between 0 and 7.

PLOTTING COORDINATES

Locating and Activating Pixels

An example of the programming statement that would determine the color status of your computer follows:

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
```

You could also use the POINT function to invert the current state of a point, as shown in the example below:

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

PSET STATEMENT

The PSET statement is used to turn on a point at a specified location on the screen.

Format 1: `PSET (X coordinate , Y coordinate) [,attribute]`
Format 2: `PSET STEP (X offset, Y offset)`

The first argument to PSET is the coordinate of the point that you wish to plot. Format 1 is the absolute form, which means you specify a point without regard to the last point referenced. An absolute point is the exact address of a pixel on the screen.

Example:

```
PSET (10,10)
0,0
. . . . . . . . . .
                  :
                  :
                  :
                  :
                  * 10,10
```

In this case PSET would turn on a dot at the location indicated by the asterick.

Suppose you had already plotted an absolute coordinate and you wanted to plot several other points relative to the last point referenced. Instead of trying to estimate the exact coordinate of your next point, you could use the second format of the PSET statement. Using the example above, if you wanted your next point to appear 10 horizontal points from 10,10 (the last point referenced) you would use format 2, as follows:

```
PSET STEP (10,0) 'offset 10 in X and 0 in Y
```

PLOTTING COORDINATES

Locating and Activating Pixels

This statement tells PSET to offset the point by 10 in X and zero in Y. Thus your next point would be turned on at the 20,10 address.

Note that 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.

Note that (0,0) is always the upper left hand corner and the bottom left corner is (0,225). It may seem strange to start numbering Y at the top, but, this is standard.

The last argument to the PSET statement allows you to specify the color you want the point to be turned on in. 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. You can use the PSET statement with a color argument to turn off points by adding a color argument that is the same as the background color as shown in the example that follows.

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
```

PLOTTING COORDINATES

Locating and Activating Pixels

PRESET STATEMENT

PRESET has an identical format 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.

Format 1: `PRESET (Xcoordinate , Y coordinate) [,attribute]`
Format 2: `PRESET STEP (X offset, Y offset)`

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 PRESET (I,I)
60 NEXT
```

Notice that this example is the same example given for PSET on Page 7.10. The only difference is in line 50, where the third parameter is not specified.

The PRESET statement defaults to the background color and causes all of the specified points to be turned off. If a color argument was added to this line, the affect 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 error message.

PLOTTING COORDINATES

Changing the Cursor Position

BRIEF

Three statements that affect the cursor are: LOCATE, CSRLIN, and POS. These statements use rows and columns as their arguments.

The LOCATE statement moves the cursor to the specified position on the Screen.

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

The CSRLIN function returns the current line (or Row) position of the cursor.

Format: `X = CSRLIN`

The POS function returns the current column position of the cursor.

Format: `POS(I)`

Details

The LOCATE statement moves the cursor to the specified position on the Screen. The last optional parameter turns the cursor on and off.

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

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 80.

PLOTTING COORDINATES

Changing the Cursor Position

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

Action:

The LOCATE statement moves the cursor to the specified position. Subsequent PRINT statements begin placing characters at this location.

Rules:

1. Any values entered outside of the row and column ranges will result in an `Illegal Function Call` error message. 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 cursor visible, position remains unchanged.
`30 LOCATE 5,1,1`	Move to line five, column one, turn cursor on.

PLOTTING COORDINATES

Changing the Cursor Position

CSRLIN AND POS FUNCTION

The CSRLIN function returns the current line (or row) position of the cursor. The POS function returns to the current column.

Format: `X = CSRLIN`

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

x = POS(I) will return the column location of the cursor. The value returned 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.
```

Unlike the coordinates of points, which start at point 0,0, the coordinates of rows and columns start at position 1,1.

CHAPTER 8 ADVANCED COLOR GRAPHICS

Using Color Graphics

BRIEF

When using color with any of the advanced graphics statements, you can specify the attribute to be used. Available colors can be one of the following eight.

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

The color statement is used to select the foreground color and background color for screen display.

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

Details

THE VIDEO BOARD

The video board can be purchased with or without color capability. The difference between a computer that has color and one that does not is for the most part in the video RAM chips that contain the information necessary for producing color. A monochrome video board has 64K of video RAM (Random Access Memory) and a color video board has at least 96K of video RAM. It is possible to upgrade a monochrome video board to color.

Note that the use of the extended character set with special H-19 graphic characters is not considered “graphics”.

As mentioned in Chapter 7, the Z-100’s video resolution is 640 by 225 (with the 25th line) — 3 bits per pixel.

ADVANCED COLOR GRAPHICS

Using Color Graphics

When storing graphics memory with PSET, PRESET or LINE you can select the "attribute" (color) from one of eight values.

It is fairly simple to produce graphics since a pixel (point) only has a value of zero or one. A zero pixel is always associated with the color black. A one pixel can associate with various intensities of white through the first argument to the color statement.

Advanced graphics extend the capabilities to manipulate the graphics mode bit map provided by the color video card.

The statements we have included in our discussion of advanced graphics are:

COLOR	PUT
LINE	GET
CIRCLE	DRAW
PAINT	

THE COLOR STATEMENT

The format of Color statement is:

COLOR [Foreground] [,Background]]

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. Those colors that contain green will display as green. Any other color will display as black. 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.

Using Color Graphics

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

Valid Colors

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

Refer to first the example shown on Page 7.8 to determine if the computer you are using has a monochrome or color video board.

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 end in a comma (,). For example, COLOR , 7, leave the background unchanged.

ADVANCED COLOR GRAPHICS

Using Color Graphics

Example:

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

Example:

```
10 CLS
20 FOR J=0 to 7
30 COLOR J,7-J: PRINT " "
40 NEXT J
```

This program will draw eight boxes in the upper left-hand corner of your screen and will fill them with the eight valid colors. Black is the color of the last box, however it is not visible on a black background.

Even though there are 8 valid colors, the range of numbers you may use to specify colors is 0 to 255. If you specify a color larger than 7, BASIC will use MOD 8 to reduce the number to its true value.

ADVANCED COLOR GRAPHICS

LINE, CIRCLE and PAINT Statements

BRIEF

Three powerful graphic statements that Z-BASIC uses to create graphic images on the screen are the LINE, CIRCLE, and PAINT statements.

The LINE statement permits the drawing of lines in absolute and relative locations on the screen. It can also be used to make boxes and filled boxes.

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

The CIRCLE statement draws an ellipse with a center and radius as specified by the arguments.

Format:
```
CIRCLE (X center,Y center),radius
        [,attribute[,start,end[,aspect]]]
```

The PAINT statement is used to fill graphic figured with the specified PAINT attribute, until it reaches the specified border attribute.

Format:
```
PAINT (X start,Y start)[,paint attribute
        [,borderattribute]]
```

Details

THE LINE STATEMENT

LINE is the most powerful of the graphic 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 a line from the last point referenced to the point (X2,Y2) in the foreground attribute. The foreground attribute is the default attribute.

ADVANCED COLOR GRAPHICS

LINE, CIRCLE and PAINT Statements

We can include a starting point also:

```
LINE (0,0)-(319,199) 'draw diagonal line down screen
LINE (0,100)-(319,100) 'draw bar across screen
```

We can append a color argument to draw the line in green, which is color 2:

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

10 CLS
20 LINE -(RND*639,RND*224),RND*7
30 GOTO 20 'draw lines forever using random attribute
```

The final argument to LINE is ",b" — box or ",bf" — filled box. The syntax indicates 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
```

or the attribute can be included:

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

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

```
10 LINE (0,0)-(0,224),1
20 LINE (0,224)-(639,224),1
30 LINE (639,224)-(639,0),1
40 LINE (639,0)-(0,0),1
```

This program uses the four corner points of the screen (as mentioned in Chapter 7) and forms a border around the screen. Using the box option of the LINE statement, the equivalent function could be performed with the following statement:

```
10 LINE (0,0)-(639,224),1,b
```

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

LINE, CIRCLE and PAINT Statements

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

STEP (X offset,Y offset), which is the relative form may be used in any of the graphic statements that reference absolute points. 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 PSET (100,100)
20 LINE STEP (20,20)-STEP (50,50)
```

In this example the PSET statement was used to turn on a point at (100,100). Then a line was drawn from the last point referenced (100,100). The STEP offset of (20,20) tells BASIC to begin the line at point (120,120). The STEP offset of (50,50) tells BASIC to end the line at (170,170).

Example:

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

ADVANCED COLOR GRAPHICS

LINE, CIRCLE and PAINT Statements

As a final example in our discussion of the LINE statement, we have included a program that creates a bar graph.

```
10 CLS
20 Y=200
30 X=50
40 XI=50
50 LINE (0,0)-(640,215),2,B  'border
60 LINE (X-5,Y)-(X-5,10)    'y axis
70 LINE (X-5,Y)-(600,Y)     'x axis
80 FOR J=1 TO 10
90 READ PT
100 YPT=100-PT
110 LINE (X,Y)-(X+20,YPT),1,BF
120 X=X+XI
130 NEXTJ
140 END
150 DATA 10, 20, 15, 25, 30, 22, 30, 60, 70, 85
```

In this program the screen is cleared, and the variables Y,X, and X1 are initialized. Program line 50 is a LINE statement with the box option included to make a green border around the graph. Program lines 60 and 70 are lines that form the y and x axis respectively.

Line 80 is the beginning of the FOR... NEXT loop that tells BASIC it will perform the following function 10 times. Line 90 tells BASIC to read the DATA statement in line 150 to determine what percent value each bar in the graph should reflect.

Line 100 determines the height each bar will be. If you were drawing a bar graph without the assistance of a computer, your zero mark would naturally start in the lower left corner. However, as we mentioned before, the zero point on the computer is in the top left corner. Thus it is necessary to change the point of orientation, which is what line 100 is actually doing.

Line 110 draws filled boxes of different lengths, reflecting the percentages in the DATA statement. Line 120 sets the distance between each bar of the graph. Line 130 ends the FOR NEXT loop, and 140 ends the program.

THE CIRCLE STATEMENT

The CIRCLE statement draws an ellipse with a center and radius as indicated by the first of its arguments.

Format:
```
CIRCLE (X center,Y center),radius
        [,attribute[,start,end[,aspect]]]
```

In the format, the X and Y are the coordinates of the center point of the ellipse. Radius is the distance from the center to the edge of the circle. Attribute is an optional argument that determines the color of the circle. The default attribute is the foreground color.

The start, end parameters are angles described in radians where 6.28 is the total amount of radians in the circle. 6.28=2*PI. PI is equal to 3.14159, which is equal to half of a circle. Figure 8.1 illustrates the angles of a circle.

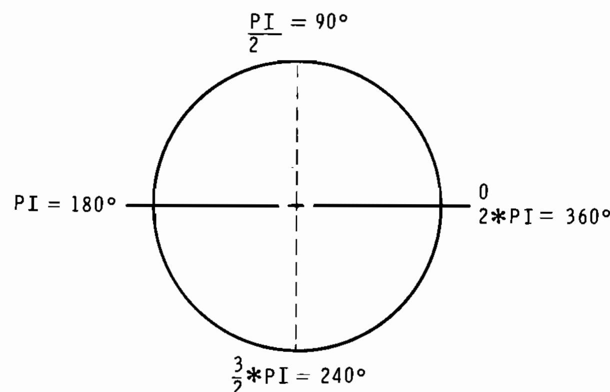


Figure 8.1.
Angles of a Circle

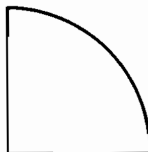
The start and end angle parameters are radian arguments between 0 and 2* PI which allow you to specify where drawing of the ellipse will begin and end.

If the start and/or end angle is negative, the ellipse will be connected to the center point with a line. The angles will be treated as if they were positive (Note that this is different than adding 2*PI). The start angle may be less than the end angle, but neither may be 0 (the equivalent of zero — a very small number — may be used in its place).

Example:

```
10 CIRCLE (100,100),50,7,-.001,-1.5707
```

This program line will draw a pie slice as illustrated below. Note that 1.5707 is half of PI.



The aspect ratio describes the ratio of the X radius to the Y radius. It determines what kind of ellipse is to be drawn. The default aspect ratio is .4843 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.

```
10 CLS
20 CIRCLE (320,110),200*RND,7,0,2*3.14159,RND
30 GOTO 20
```

This example clears the screen and draws continuous circles on the screen. The radius is 200 * a random number. The start angle is 0, and the end angle is 2*PI or 360°. The aspect ratio is a random number from 1-0.

NOTE: Make sure your background color is not 7 before running this program, otherwise the circles will not be visible.

Example:

```
10 RAD=5
20 CLS
30 CIRCLE (320,110),RAD,7,0,2*3.14159,
        (RND+RND+RND+RND)/4
40 RAD=RAD+7.5
50 IF RAD>175 THEN END ELSE 30
```

This example is similar to the one above except instead of a random radius, the radius is assigned the value of 5 and incremented by 7.5 each time an ellipse is drawn. When the radius becomes larger than 175, the program ends.

Notice in line 30 four RND functions are added together and then divided by 4 to get an average random number. This helps decrease the variance of the aspect ratio. More often than not the aspect ratio will be close to .5 since RND is a number between 0 and 1.

For our last example of the CIRCLE statement, this program will draw two cone shaped figures on the screen.

```
10 CLS: D=1
20 X=78: Y=112
30 RAD=73: ASP=.9
40 AG1=0
50 AG2 =2*3.14159
60 C=7
70 CIRCLE (X,Y),RAD,C,AG1,AG2,ASP
80 RAD=RAD-3*D
90 X=X+10
100 IF RAD<2 THEN D=-D:
    GOTO 80 ELSE IF RAD>73 AND D=-1 THEN 120 ELSE 70
110 END
```

The program begins by clearing the screen and assigning the variable D a value of 1, X=78, Y=112, radius=73, and the aspect ratio = .9. Angle 1=0 and angle 2=2*PI. The foreground color is number 7 which is white.

Line 70 tells BASIC to draw an ellipse using the previously assigned variables. Line 80 says each time an ellipse is drawn, decrease the radius by −3. Decreasing the radius by −3 means the ellipse will get smaller and smaller.

Line 90 tells BASIC to increment the value of X by 10 for every ellipse drawn. If you could connect the center point of each ellipse you would find they form a straight line.

Line 100 says if the radius is less than two, then D becomes negative. When −D is multiplied by −1 and added to RAD, the result is a positive number and the ellipses begin to get larger as the radius increases.

Line 100 then tells BASIC to continue drawing ellipses until the radius is larger than 73 and D is equal to −1.

THE PAINT STATEMENT

The PAINT statement will fill in any graphic figure with the attribute you specify until it reaches the specified border attribute of that figure. If no paint attribute is given, PAINT will default to the foreground attribute. If the border attribute is not given, it defaults to the PAINT attribute.

Format:
```
PAINT (X start,Y start)[,paint attribute
           [,borderattribute]]
```

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

```
10 CLS
20 CIRCLE (320,112),100,1
30 PAINT (300,100),2,1
```

This example draws a circle with a center point of (320,112) and a radius of 100, in the color blue. A point within that circle is selected (300,100). The circle is then painted with the color green from that point, until it reaches the blue border.

If the border attribute is not equal to the foreground attribute, (the color the figure is drawn in) PAINT will never see the border and will fill the entire screen with the PAINT attribute.

A problem that commonly occurs when using the PAINT statement is "holes" in the border that permit the PAINT to seep out and place a color in undesirable places. You must be sure your coordinates are correct and all of the points that make up your boundaries are included in your graphic statement.

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

LINE, CIRCLE and PAINT Statements

The PAINT statement can be used with other graphic statements.

Example:

```
10 CLS: LINE (0,0)-(100,200),4,B
20 PRESET (100,100)
30 LINE (200,0)-(300,200),4,B
40 LINE (100,90)-(200,90),4
50 PRESET (200,100)
60 LINE (100,110)-(200,110),4
70 PAINT (1,1),1,4
```

This example will draw two retangular boxes down the screen and a smaller rectangle in the middle. Then it paints the boxes blue until it gets to the red border. PRESET is used to turn off a point within the graphic to begin painting from.

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.

ADVANCED COLOR GRAPHICS

GET, PUT, and DRAW Statements

BRIEF

The DRAW statement combines many of the capabilities of the other graphic statements into a graphics macro language that permits the drawing of graphic images on the screen.

Format: `DRAW <"string expression">`

After your graphic image is drawn you may want to use the GET, PUT statements to transfer the image to and from the screen.

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

The GET and PUT statements are also used for computer animation and for other special effects involving moving objects on the screen.

Details

THE DRAW STATEMENT

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 ®. A GML command is a single character within a string, optionally followed by one or more arguments.

Format: `DRAW <"string expression">`

The DRAW statement can be assigned to a string expression, in the following manner:

```
10 A$="U2L2D2R2"
20 DRAW A$
```

If a DRAW statement is assigned in this manner, you could use this movement sequence in another place in your program without having to input the entire DRAW statement.

The DRAW statement, when used with other graphic statements will begin drawing at the last point referenced. When used with the CIRCLE statement, it will begin drawing at the center point of the circle.

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 referenced 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

These commands move one unit if no argument is supplied. The number of points (n) always follows the command.

Example:

```
10 CLS
20 LINE (100,0)-(100,100) 'draw a line
30 B$="H10"
40 DRAW B$ 'draw a diagonal line up and left 5 points
RUN
```

ADVANCED COLOR GRAPHICS

GET, PUT, and DRAW Statements

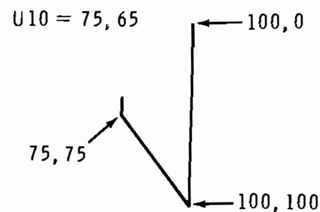
Absolute and Relative Moves

M <X,Y> Move to an absolute or relative address. As in other graphic statements, the DRAW statements can be used in absolute and relative forms. Relativity is indicated in the following manner:

If X is preceded by a "+" or "-", X and Y are added to the current graphics position, and connected to the current position with a line. Otherwise, a line is drawn to point X,Y from the current position.

Example:

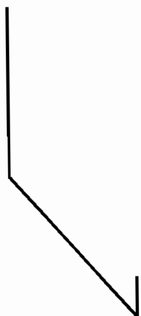
```
10 CLS
20 LINE (100,0)-(100,100) 'draw a line
30 B$="M75,75 U10"
40 DRAW B$
```



This example tells BASIC to draw a line from (100,0) to (100,100) and from that point (100,100) draw a line to the absolute address of (75,75) then draw a line up 10 points (75,65). If you inserted a "+" sign in line 30:

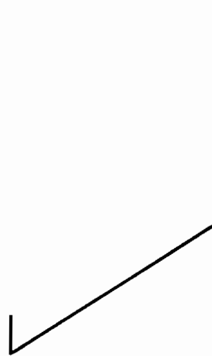
30 B$="M+75,75 U10"

the second line would be added to the current graphic position. The second line would be drawn from (100,100) to (175,175) and the last point would be at point (175,165). If you inserted a " – " sign in line 30:



30 B$="M-75,75 U10

BASIC would draw a line similar to the one above except this line would be in another direction. The second line would be drawn from (100,100) to (25,175) and the last point would be at point (25,165). The "+" and the "-" indicate relative starting points.

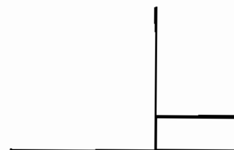


A[<n>] Set angle n. n may range from zero to three, where zero is zero degrees, one is 90, two is 180, and three is 270. Figures rotated 90 or 270 degrees are scaled so that they will appear the same size as with zero or 180 degrees on a monitor screen with the standard aspect ratio of 31/64. In the following example we will demonstrate how the angle command is used to rotate a box to different positions in relationship to the reference point (100,100).

Example:

```
10 CLS
20 LINE (100,0)-(100,100)
25 LINE (0,100)-(100,100)
30 B$="U10 R50 D10 L50"
40 DRAW B$
```

This example draws a X and Y axis, and using 100,100 as the starting point, draws a small box just to the right of the reference point as shown below.



ADVANCED COLOR GRAPHICS

GET, PUT, and DRAW Statements

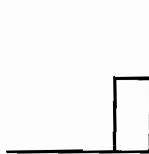
If an angle command A0 was added to line 30,

30 B$="A0 U10 R50 D10 L50"

the box would appear in the same position it was in the previous example, because zero is the default angle. If A1 was substituted in line 30,

30 B$="A1 U10 R50 D10 L50"

the box would be rotated 90 degrees and appear as shown below.

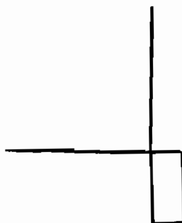


If A2 was substituted in line 30,

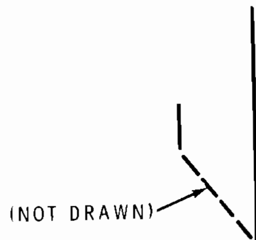
30 B$="A2 U10 R50 D10 L50"

the box would be rotated 180 degrees and appear as shown below.

Finally, A3 would cause the box to be rotated 270 degrees and appear as shown below.



PREFIX COMMANDS



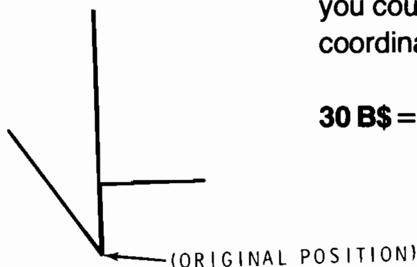
B Move but don't plot any points. The B command permits you to move to a different location without plotting any points. If you inserted a B in front of a movement command in line 30:

30 B$ = "BM75,75U10"

The line created by M75,75 would not be drawn.

N Move and return to original position. If the N prefix were added in line 30 you could move back to the original position without specifying the original coordinates.

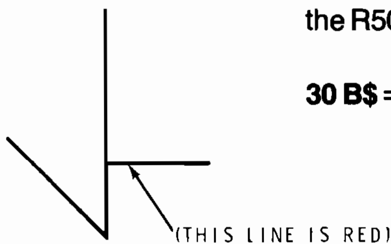
30 B$ = "NM75,75U10R50"



The M75,75 was drawn and the cursor returned to the original position (100,100) and then moved up ten. R50 (right 50) was included to demonstrate where the next line would be drawn from.

C[<N>] Set the color. Using the same example, C4 was inserted before the R50 to set that line in the color red.

30 B$ = "NM75,75U10C4R50"



Note: Remember to place the prefix commands in front of the movement commands, otherwise you will receive an Illegal function call error message. For example,

30 B$ = "NM75,75U10RC450"

would yield an error because the color prefix is in the middle of R50.

ADVANCED COLOR GRAPHICS

GET, PUT, and DRAW Statements

As you can see from these examples the angle command rotates the figure in 90 degree increments using the last point referenced as a starting point. Remember to change your angle back to the default angle if you wish to continue programming after you practice using this command. BASIC remembers the last angle used and this will affect any design that follows.

S<n>

Set scale factor. n may range from zero to 255. The scale command is used to increase or decrease the size of a figure by the scale factor specified. The scale factor multiplied by the distances given with U,D,L,R or relative M commands is used to get the actual distance traveled.

A scale factor of S0 returns the orginal size of the figure. If you wanted the figure to be smaller than its orginal size, you would select a size from one-three. S4 will also return the original size, and anything larger than S4 will return a larger figure.

This program draws A$ 35 times, each time incrementing the scale factor by 1.

As with the angle command, the scale command must also be returned to S0 before programming continues.

Example:

```
10 CLS
20 PSET (0,200),7
30 A$="U20R20D20L20"
40 FOR J=1 to 35
50 DRAW "S"+STR$(J)
60 DRAW A$
70 NEXT J
```

X <string;> Execute substring (not supported by BASIC compiler). This 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).

Example:

```
10 CLS
20 PSET (20,20),7
30 A$="U20R20D20L20"
50 DRAW "S1XA$;S10XA$;S20XA$;S50XA$;S100XA$;"
```

This program executes the substring A$ in 5 different sizes.

ADVANCED COLOR GRAPHICS

GET, PUT, and DRAW Statements

GET AND PUT STATEMENTS

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

The GET statement transfers the screen image into an array. The image is contained within the boundaries of a rectangle defined by the specified points. The rectangle is defined the same way as the rectangle drawn by the LINE statement using the ",B" option. See Page 8.6.

The array is simply used as a place to hold the image and can be any type except string. It must be dimensioned large enough to hold the entire image.

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 storage format in an 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 formula used to determine required array size in bytes is:

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

WHERE: bits per pixel is 3

X = number of columns to be stored

Y = number of rows to be stored

ADVANCED COLOR GRAPHICS

GET, PUT, and DRAW Statements

The bytes per element of an array are:

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

Following is a step-by-step procedure for using the GET PUT statements.

1. Create a graphic image using the DRAW statement and or any of the other graphic statements.

2. Calculate the size of the array using the formula mentioned on the preceding page.

3. GET the image and store it into an array.

4. PUT the image on the screen in a new location.

5. RUN the program to see the image move to the new location.

```
10 CLS
20 PRINT: PRINT "AB"
30 LINE(0,0)-(20,20),3,B
40 DIM A#(25)
45 FOR J=1 TO 200:NEXT
50 GET (0,0)-(20,20),A#
55 FOR J=1 TO 200:NEXT
60 CLS
70 PUT (25,25),A#
```

This program clears the screen, prints a blank line and then prints “AB”. In line 30 a cyan box is drawn, 21 by 21 pixels in size. The value of 21 is used in the formula, not 20. This is because coordinates start at the 0,0 address. The rectangle is (line (0,0)-(20,20)) 21 by 21 pixels. In this example both X and Y are 21.

After the graphic image is drawn, you must determine the size and type of array it is to be stored in. Arrays can be of three types, integer, single-precision or double-precision. In this program, double-precision is used as indicated by the pound sign (#).

ADVANCED COLOR GRAPHICS

GET, PUT, and DRAW Statements

To determine the size the array should be, use the formula repeated below:

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

4+INT((21+7)/8)*3*21 = 193

The result of this calculation indicates you must have an array large enough to hole 193 bytes. The next question is, how many bytes per element will be in this array? If you declared the array to be integer, (%) you would compute 193/2. If the array were declared single-precision, (!) you would compute 193/4. This program declares the array double-precision, (#) thus you compute 193/8 which is 24.125.

The result of this division should be rounded up to the next largest whole number. In this case the array is dimensioned to 25 bytes, (see line 40 on the previous page).

Line 40 dimensions the array to 25 bytes per element.

Line 45 and 55 are pauses included so that you will have time to see what is happening.

Line 50 uses the GET statement to get the objects found within the retangular boundaries and records the image in memory.

Line 60 clears the screen. Line 70 GETs the image and places it on the screen at location (25,25).

ACTION VERBS

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 (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 **twice**, the background is restored unchanged. This allows you to move an object around the screen without obliterating the background.

The default action verb 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, PUT the object(s) at the new location.

Movement done this way will leave the background unchanged. Flicker can be reduced 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.

ADVANCED COLOR GRAPHICS

GET, PUT, and DRAW Statements

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 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. This method may be desirable since only one PUT is required to move an object (although you must PUT a larger image).

It is possible to examine the X and Y dimensions and even the data itself 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. Integers are stored low byte first, then high byte, but the data is transferred high byte first (leftmost) and then low byte.

The contents of the array after a GET will be meaningless when interpreted directly unless the array is of type integer and you design a special program that allows you to examine the contents of an array. We have included such a program on the next page for the convenience of the experienced user. For additional information on Arrays see Page 5.13.

```
10 '    Character Image display program

20 CLEAR 100 '                          Clear some string space

30 INPUT"Character :",C$ '              Get the character to be imaged
35 GOSUB 480'                           Sets up binary conversion string table

40 INPUT"Color Number <7>:",R '         Get the characters'  color
50 IF R<1 OR R>6
     THEN R=7 '                         Default color when none specified
60 INPUT"Positive or Negative <P>:",I$ '
                                        Get Image transformation
61 IF I$="" OR I$="P" OR I$="p"
      THEN I=1
      ELSE I=0 '                        Default no-transformation

65 INPUT"Mask 0,1,None <None>",M$ '     Masking bit for string edit
66 IF M$<>"0" AND M$<>"1"
      THEN M$="" '                      Default Masking bit

80 CLS '                                Clear the Screen

90 COLOR R '                            Set the character's color
100 DIM P00%(19),P01%(19),P02$(5) '     Set aside some array space

110 PRINT C$ '                          Print the character in the upper
                                        left corner of the screen
115 COLOR 7 '                           Change color to seven

120 GET(7,8)-(0,0),P00% '               Copy the image of the character into
                                        the image array

130 IF I=0
      THEN PUT(0,0),P00%,PRESET:
           GET(0,0)-(7,8),P00% '        If a negative image was requested,
                                        copy the negative image into the array
140 PRINT '                             Display the images' X and Y coordinates
150 PRINT "Pixels","Scan"
160 PRINT "Across","Lines"
170 PRINT P00%(0)/4,P00%(1)
180 PRINT '

190 PRINT "Blue","Red","Green" '        Display headings for color planes
200 PRINT '
```

ADVANCED COLOR GRAPHICS

GET, PUT, and Draw Statements

```
210 FOR Y=2 TO 16 STEP 3 '            Set up loop for store/display
220     X=Y
230     GOSUB 300 '                   Go store image

240     GOSUB 380 '                   Edit image

250     GOSUB 440 '                   Go display image

260 NEXT Y '                          Store/display loop-back

270 END '                             Termination of program

280 'Subroutines

290 '            Store image into String array

300 XDEC=P00%(X):GOSUB 570:P02$(0)=RIGHT$(STRING$(16,48)+BIN$,8)
310 P02$(1)=LEFT$(RIGHT$(STRING$(16,48)+BIN$,16),8)
320 XDEC=P00%(X+1):GOSUB 570:P02$(2)=RIGHT$(STRING$(16,48)+BIN$,8)
330 P02$(3)=LEFT$(RIGHT$(STRING$(16,48)+BIN$,16),8)
340 XDEC=P00%(X+2):GOSUB 570:P02$(4)=RIGHT$(STRING$(16,48)+BIN$,8)
350 P02$(5)=LEFT$(RIGHT$(STRING$(16,48)=BIN$,16),8)
360 RETURN:'     END OF STORE IMAGE SUBROUTINE
370'             Edit String array
380 IF M$=""
        THEN 420
385 FOR J=0 TO 5
390     K=INSTR(P02$(J),M$)
400     IF K<>0
             THEN MID$(P02$(J),K,1)=" ":
             GOTO 390
410 NEXT J
420 RETURN '     End of Edit String subroutine
430 '            Display String array
440 PRINT P02$(0),P02$(1),P02$(2)
450 IF X+2=16
        THEN 470
460 PRINT P02$(3),P02$(4),P02$(5)
470 RETURN '     End of Display String array subroutine
480 B$(0)="000"
490 B$(1)="001"
500 B$(2)="010"
510 B$(3)="011"
520 B$(4)="100"
530 B$(5)="101"
540 B$(6)="110"
550 B$(7)="111"
560 RETURN
570 DPC$=OCT$(XDEC):BIN$="":FOR YCOUNT=1 TO LEN(DPC$): 'Convert decimal value to binary strin
580 Q99=VAL(MID$(DPC$,YCOUNT,1)):BIN$=BIN$+B$(Q99):NEXT YCOUNT
590 RETURN
```

ADVANCED COLOR GRAPHICS

GET, PUT, and Draw Statements

This is a program that draws a little stickman doing acrobatics. He springs from platform to platform while doing jumping-jacks. Many of the statements we have discussed in Z-BASIC are used in this program. The program is documented with remark statements. Study the program carefully, input the program on your computer, and then try to make some modifications. You will find that GET and PUT can be used to make very creative graphic displays.

```
10 DIM A# (16) '                       Set up array for 1st stick figure
20 DIM B# (16) '                       Set up array for 2nd stick figure
30 CLS '                               Clear the Screen
40 CIRCLE (5,5),5 '                    Draw 'head' of stick figure
50 DRAW"BM5,9D2NL5NR5D3NG5NF5" '       Draw 'body' with arms & legs extended
60 GET(0,0)-(10,19),A# '               Store image of 1st figure
70 PRINT:PRINT:PRINT:
   PRINT"1st stick figure is set up" ' Print a few blank lines, then the message
80 FOR I=1 TO 777:NEXT I:CLS '         Pause to say figure is set up, then CLS
90 CIRCLE (5,5),5 '                    Now draw 'head' of 2nd figure
100 DRAW"BM5,9D2NM-5,-2NM+5,-2D3NM+2,5NM-2,5" '
                                       Draw 'body' in a 'jumping jacks' position
110 GET(0,0)-(10,19),B# '              Store image of 2nd stick figure
120 PRINT:PRINT:PRINT:
    PRINT"2nd stick figure is set up" 'Print a few blanks, then message
130 FOR I= 1 TO 777:NEXT I:CLS '       Give some time to read message, then CLS
140 LINE (30,45)-(100,0),6,B '         Draw border of acrobatics area
150 LINE (55,20)-(75,20) '             Draw top spring-board
160 LINE (80,45)-(100,45) '            Draw right spring-board
170 LINE (30,45)-(50,45) '             Draw left spring-board
180 FOR Y=25 TO 0 STEP -5
190 IF Y MOD 10 = 0 THEN GOSUB 310 ELSE GOSUB 350
200 NEXT Y '                           These lines make figure go left & up
210 FOR Y=0 TO -25 STEP -5
220 IF Y MOD 10 = 0 THEN GOSUB 310 ELSE GOSUB 350
230 NEXT Y '                           These lines make figure go left & down
240 FOR Y=-25 TO 0 STEP 5
250 IF Y MOD 10 = 0 THEN GOSUB 310 ELSE GOSUB 350
260 NEXT Y '                           These lines make figure go right & up
270 FOR Y=0 TO 25 STEP 5
280 IF Y MOD 10 = 0 THEN GOSUB 310 ELSE GOSUB 350
290 NEXT Y '                           These lines make figure go right & down
300 GOTO 180 '                         Program will end if CTRL & C is pressed
310 PUT(60+Y,ABS(Y)),A#,XOR '          Subroutine to put 1st figure on screen
320 FOR I=1 TO 75:NEXT I '                     pause for a short time
330 PUT(60+Y,ABS(Y)),A#,XOR '                  and then erase 1st figure
340 RETURN
350 PUT(60+Y,ABS(Y)),B#,XOR '          Subroutine to put 2nd figure on screen
360 FOR I=1 TO 75:NEXT I '                     pause for a short time
370 PUT(60+Y,ABS(Y)),B#,XOR '                  and then erase 2nd figure
380 RETURN
```

ADVANCED COLOR GRAPHICS

Z-BASIC Summary Program

BRIEF

Following is a summary program that uses Z-BASIC graphic commands. It is designed to demonstrate the ease and flexibilty of Z-BASIC. The program segments are fairly simple and you should have no problems determining what is actually going on. If you do have problems, refer to the appropriate sections in the manual and review statement(s) that are causing confusion.

The program is divided into two parts. DEMO I demonstrates the statements relative to plotting coordinates. DEMO II demonstrates the commands relative to advanced graphics. Again, it will probably be most helpful if you input the program, note the visual effect of the program segments, and then try some modifications of your own.

Details

```
1 ' ZBASIC Demo I (c)1982 Zenith Data Systems
10 DEFINT I-N:RANDOMIZE TIME/DATE
20 CLS
30 FOR J= 0 TO 7 : PSET (0,0),J: IF POINT (0,0) <> J THEN
                                  COLOR.COMPUTER=0
                                  ELSE NEXT J: COLOR.COMPUTER=1
40 FOR J=0 TO 7:COLORS(J)=J:NEXT J ' COLORS CONTAINS AVAILABLE COLOR ATTRIB.
50 IF COLOR.COMPUTER=0 THEN FOR J=1 TO 7:COLORS(J)=7:NEXT J
              ' BLACK AND LOTS OF WHITE
100 CLS:PRINT"This is a demonstration of the PSET command....";:GOSUB 10000
110 CLS:FOR J=1 TO 150: PSET (RND*640,RND*215),COLORS((RND*7)+1):NEXT J
120 LOCATE 8,10:PRINT"Space......":LOCATE 9,10:PRINT"The final frontier..."
130 GOSUB 10000
200 CLS:PRINT"This is a demonstration of the PSET and PRESET commands.":
        GOSUB 10000:CLS
210 DIM A(150,3)
220 FOR J=1 TO 150 :A(J,1)=RND*640:A(J,2)=RND*215:A(J,3)=RND*7+1
230 PSET (A(J,1),A(J,2)),COLORS(A(J,3)): NEXT J
240 FOR J=1 TO 150: PRESET (A(J,1),A(J,2)),COLORS(0):NEXT J
250 ERASE A
300 CLS:PRINT"Here is a demonstration of the POINT command w/ PSET.":
        GOSUB 10000:CLS
310 FOR K=1 TO 5:IF K=1 THEN
        FOR J=100 TO 200: PSET (J,100),COLORS(4):
                          PSET (J,200),COLORS(4):
                          PSET (100,J),COLORS(4):
                           PSET (200,J),COLORS(4): NEXT J
320 X=101+(2*K):Y=101
```

```
330 IF POINT (X,Y)=COLORS(4) THEN LOCATE K,20:PRINT"I hit the
wall!":GOTO 390
340 PSET (X,Y),COLORS(K):Y=Y+1:X=X+1
350 GOTO 330
390 NEXT K:GOSUB 10000
400 CLS:PRINT"Here is a demonstration of the CSRLIN and POS com-
mands."
405 GOSUB 10000: CLS
410 CLS: FOR K=1 TO 5:ROW=INT(RND*23)+1:COL=INT(79*RND)+1
420 LOCATE ROW,COL:PRINT"*";:
        NEWCOL=POS(0)-1:
        PRINT:PRINT"The star is at row";ROW;"and column";NEWCOL
430 GOSUB 10000:CLS:NEXT K
500 CLS
510 PRINT"Would you like to continue on with DEMO II";
520 INPUT A$:A$=LEFT$(A$,1)
530 IF A$="Y" OR A$="y" THEN RUN"DEMOII"
540 IF A$="n" OR A$="N" THEN CLS:PRINT"Thanks for watching.":END
550 PRINT"Please answer Yes or NO!":GOTO 510
10000 FOR J=1 TO 1000:TEMP.RND=RND:
          IF INKEY$=CHR$(13) THEN RETURN
                                    ELSE NEXT J: RETURN
```

ADVANCED COLOR GRAPHICS

Z-BASIC Summary Program

```
1' ZBASIC Demo II (c)1982 Zenith Data Systems
10 DEFINT I-N:RANDOMIZE TIME/DATE
20 CLS
30 FOR J= 0 TO 7 : PSET (0,0),J: IF POINT (0,0) <> J THEN
                              COLOR.COMPUTER=0
                              ELSE NEXT J: COLOR.COMPUTER=1
40 FOR J=0 TO 7:COLORS(J)=J:NEXT J ' COLORS CONTAINS AVAILABLE COLOR ATTRIB.
50 IF COLOR.COMPUTER=0 THEN FOR J=1 TO 7:COLORS(J):=7:NEXT J:
               ' BLACK AND LOTS OF WHITE
100 CLS:PRINT"This is an example of the COLOR command."
110 IF COLOR.COMPUTER=0 THEN
        PRINT"Sorry, this isn't very clear on a black and white system."
120 GOSUB 10000:CLS:BG=7:FOR J=0 TO 7:COLOR COLORS(J),COLORS(BG)
130 PRINT "This line is in color #";COLORS(J)
        ;"with a background color #";COLORS(BG):
        BG=BG-1:NEXT J:GOSUB 10000
200 CLS:PRINT"The following are examples of the four LINE usages.":GOSUB 10000
210 CLS:
        FOR J=1 TO 10: LINE -(RND*640,RND*215),COLORS(RND*7):NEXT J:GOSUB 10000
220 CLS:FOR J=1 TO 10:
        LINE (RND*640,RND*215)-(RND*640,RND*215),COLORS(RND*7):NEXT J:
        GOSUB 10000
230 CLS:
        FOR J= 1 TO 10:LINE (RND*640,RND*215)-(RND*640,RND*215),COLORS(RND*7),B:
        NEXT J:GOSUB 10000
240 CLS:
        FOR J= 1 TO 10:LINE (RND*640,RND*215)-(RND*640,RND*215),COLORS(RND*7),BF:
        NEXT J:GOSUB 10000
300 CLS:PRINT"The following are examples of the four CIRCLE usages.":
                                                                 GOSUB 10000
310 CLS:CIRCLE (320,110),100,COLORS(RND*6)+1:GOSUB 10000
320 CLS:CIRCLE (320,110),100,COLORS(RND*6)+1,-2*3.14159,-RND*2*3:GOSUB 10000
330 CLS:CIRCLE (320,110),100,COLORS(RND*6)+1, , ,.1:GOSUB 10000
340 CLS:CIRCLE (320,110),100,COLORS(RND*6)+1,-2*3.14158,-RND*2*3.1 ,.1:GOSUB 10000
```

```
400 CLS:PRINT"The following are examples of GET and PUT":GOSUB 10000
405 DIM H%(13),E%(13),L%(13),O%(13):'Dimension the arrays used for GET & PUT
410 CLS:PRINT"H":GET(2,1)-(6,7),H%:' Get H
415 CLS:PRINT"e":GET(2,1)-(6,7),E%:' Get e
420 CLS:PRINT"l":GET(2,1)-(6,7),L%:' Get l
425 CLS:PRINT"o":GET(2,1)-(6,7),O%:' Get o
430 CLS:FOR Z=1 TO 15:'            Print Hello (slanted) 15 times
435 X=RND*600:Y=RND*200
440 CLS:PUT(X,Y),H%:PUT(X+7,Y+2),E%:PUT(X+14,Y+4),L%
445 PUT(X+21,Y+6),L%:PUT(X+28,Y+8),O%
450 GOSUB 10000: NEXT Z
500 CLS:PRINT"The following are examples of the PAINT command.":GOSUB 10000
510 CLS: CIRCLE (320,110),50,COLORS(7):PAINT (320,110),COLORS(5),COLORS(7):
        GOSUB 10000
520 CLS: LINE (100,100)-(200,200),COLORS(7),B:
        PAINT (101,101),COLORS(2),COLORS(7):GOSUB 10000
530 CLS:LINE (0,0)-(600,20),COLORS(7),B:LINE -(590,20),COLORS(0):
        LINE (590,20)-(590,200),COLORS(7):LINE(600,20)-(600,200),COLORS(7):
        LINE (0,200)-(600,215),COLORS(7),B:LINE (599,200)-(590,200),COLORS(0)
540 LINE (0,200)-(300,20),COLORS(7):LINE (10,200)-(310,20),COLORS(7):
        LINE (1,201)-(10,200),COLORS(0):LINE (300,20)-(309,20),COLORS(0)
550 PAINT (1,1),COLORS(5),COLORS(7):GOSUB 10000

600 CLS:PRINT"Following is an example of the DRAW statement":GOSUB 10000
610 DRAW"A0S0BM 320,112" 'Sets pointer to normal
620 DOR=40' Size of door
630 ROOF$="E50R120F50"
640 LSIDE$="BL100U65XROOF$;" ' Left side of house
650 DRAW"BM+100,23D65U20" ' Right side of house
660 DRAW"BM300,200U=DOR;R=DOR;D=DOR;L=DOR;XLSIDE$;"
670 CIRCLE(305,185),2,7' Doornob
680 DRAW"BM234,145D14L12U15R12" ' Left window
690 DRAW"BM390,145 D15L12U15R12" ' Right window
700 DRAW"BM396,110U50L45D25" ' Chimney
710 PRINT "HOME SWEET HOME"
720 LINE (0,0)-(639,224),1,B ' Border
9999 END
10000 FOR J=1 TO 1000:TEMP.RND=RND:
        IF INKEY$=CHR$(13) THEN RETURN
                          ELSE NEXT J: RETURN
```

CHAPTER 9 BASIC LANGUAGE SUMMARY

Commands

BRIEF

The command statements used in BASIC are listed below. When entered, these commands will execute immediately. These commands are often used in the direct mode. However, with the exception of the CONT command, they may also be used within a program.

Details

COMMAND	DESCRIPTION
AUTO	Enables automatic line numbering.
BLOAD	Loads machine language programs into memory.
BSAVE	Saves machine language programs to the specified device.
CLEAR	Sets all numeric variables to zero and all string variables to null.
CONT	Continues program execution.
DELETE	Deletes program lines from memory.
EDIT	Displays the specified line(s) and positions the cursor at the first digit of the line number.
FILES	Displays the names of the files residing on the disk.
KILL	Erases specified disk file.
LIST	Displays all or part of the program currently in memory.
LLIST	Lists all or part of the program in memory on the line printer.

BASIC LANGUAGE SUMMARY

Commands

COMMAND	DESCRIPTION
LOAD	Loads a file from the disk into memory.
MERGE	Merges an ASCII disk program file into the program currently in memory.
NAME	Renames a disk file.
NEW	Deletes the program currently in memory and clears all variables.
RENUM	Renumbers program lines.
RESET	Enables you to exchange a new disk for the disk in the current drive.
RUN	Executes the program currently in memory.
SAVE	Writes to disk the program currently in memory.
SYSTEM	Permits you to exit Z-BASIC and return to Z-DOS.
TRON/TROFF	Turns trace on and off.

BASIC LANGUAGE SUMMARY

Statements

BRIEF

The statements available in BASIC can be divided into five functional groups: Data type definition, Assignment and Allocation, Control, Non I/O, and I/O. The following list of BASIC statements are arranged by function.

Details

DATA TYPE DEFINITION STATEMENTS

A DEF statement declares that the variable name beginning with a certain range of letters is of the specified data type. If no data type definition statements are encountered, BASIC assumes all variables without declaration characters are single-precision variables.

DEFDBL	Declares variable as double-precision.
DEFINT	Declares variable as an integer.
DEFSNG	Declares variable as single-precision.
DEFSTR	Declares variable as string data type.

ASSIGNMENT AND ALLOCATION STATEMENTS

Assignment and allocation statements are used to assign values to variables and allocate the required storage space.

DIM	Sets up the maximum values for array variables and allocates storage accordingly.
ERASE	Removes arrays from a program.
LET	Assigns value to a variable.
OPTION BASE	Specifies minimum value for array subscript.
REM	Allows explanatory remarks to be inserted in a program.
SWAP	Exchanges variable values.

BASIC LANGUAGE SUMMARY

Statements

CONTROL STATEMENTS

Two types of control statements are available in Z-BASIC. One type affects the sequence of execution, and the other type is used for conditional execution.

Sequence of Execution

The sequence of execution statements are used to alter the sequence in which the lines of a program are executed. Normally, execution begins with the lowest numbered line and continues sequentially, until the highest numbered line is reached. The sequence of execution statements allow the programmer to execute the lines in any sequence that the program logic dictates.

END	Terminates program execution.
FOR/NEXT	Allows a series of instructions to be performed in a loop a given number of times.
GOSUB/RETURN	Branches to and returns from a subroutine.
GOTO	Branches unconditionally to the specified line number.
ON COM GOSUB	Enables a trap routine for communications device.
ON ERROR GOTO	Enables an error trap routine at the specified line number.
ON/GOTO and ON/GOSUB	Evaluates an expression and branches to one of several specified line numbers.
ON KEY GOSUB	Enables a trap routine for a specified key.
RESUME	Returns from an error trap routine.
RETURN	Returns from subroutine.
STOP	Terminates program execution and returns to BASIC command mode.
WAIT	Suspends program execution while monitoring the status of an input port.

CONDITIONAL EXECUTION STATEMENTS

Conditional Execution

The conditional execution statements are used to optionally execute a statement or series of statements. The statement(s) will be executed if a certain condition is met.

IF/THEN/ELSE	Makes a decision regarding program flow based on the result returned by an expression.
WHILE/WEND	Executes a series of statements in a loop as long as the condition is true.

NON-I/O STATEMENTS

CALL	Calls an assembly language subroutine.
CHAIN	Loads a program and passes current variables to it.
COM ON/OFF	Enables and disables the trapping of communications activity.
COMMON	Passes variables to a CHAINed program.
DATA	Stores numeric and string constants.
DATE	Returns an integer value representing the day of the year.
DATE$	Sets or retrieves the current date.
DEF FN	Defines numeric or string functions.
DEF SEG	Defines current segment of memory.
DEF USR	Defines starting address for machine language subroutine.
ERROR	Simulates the occurance of an error.
KEY	Allows function keys to be designated “soft keys”.
KEY LIST	Displays soft key assignments currently in effect.

BASIC LANGUAGE SUMMARY

Statements

KEY ON/OFF	Turns soft key display on or off.
LOCATE	Moves the cursor to the specified position on the screen.
MID$	Replaces a portion of one string with another string.
NULL	Sets the number of nulls to be printed at the end of each line.
OPEN COM	Allocates a buffer for I/O.
RANDOMIZE	Reseeds the random number generator.
READ	Reads data into specified variables from a DATA statement.
RESTORE	Resets DATA pointer so that data may be reread.
TIME	Returns an integer value representing the time in seconds.
TIME$	Sets or retrieves the current time.

I/O STATEMENTS

BEEP	Sounds the speaker.
CLOSE	Concludes I/O to a disk file.
CLS	Clears the screen.
FIELD	Defines fields in a random file buffer.
GET	Reads a record from a random disk file into a random buffer.
INPUT	Allows input from the keyboard during program execution.
INPUT#	Reads data items from a sequential file.

BASIC LANGUAGE SUMMARY

Statements

LINE INPUT	Allows input of an entire line,(up to 255 characters) to a string variable without the use of delimiters.
LINE INPUT#	Reads an entire line from a file.
LPRINT	Prints data on the line printer.
LPRINT USING	Prints data on the printer using the format specified by string.
LSET	Left-justifies a string in a field.
OPEN	Allows I/O to a disk file.
OUT	Sends a byte to a machine output port.
PRINT	Displays data on the screen.
PRINT USING	Displays data using the specified format.
PRINT#	Writes data to a sequential file.
PRINT# USING	Writes data to a sequential file using specified format.
PUT	Writes data from a random file buffer to disk file.
RSET	Right-justifies a string in a field.
WRITE	Outputs data on the screen.
WRITE#	Outputs data to a file.

BASIC LANGUAGE SUMMARY

Functions

BRIEF

Z-BASIC provides a full set of intrinsic functions for use in your programs. One group of functions is the arithmetic functions. These functions are referenced by a symbolic name. When they are invoked, they return a single value which can be either an integer or single-precision data type.

Other functions called mathematical functions are not intrinsic to BASIC, but can be calculated when necessary with the formulas provided in Appendix D.

Another category of functions is the string functions which allow you to build strings, manipulate strings, convert strings, and form substrings.

Additionally, there are special functions available for enhanced programming flexibility.

Details

ARITHMETIC FUNCTIONS

ABS	Returns the absolute value.
ATN	Returns the arctangent.
CDBL	Converts to double-precision.
CINT	Converts to an integer.
COS	Returns the cosine in radians.
CSNG	Converts to single-precision.
EXP	Calculates the exponential value.
FIX	Truncates the decimal part of a specified argument.
INT	Returns the largest integer <= the variable.

LOG	Returns the natural logrithm.
RND	Returns random number between 0 and 1.
SGN	Returns the sign (+,- or 0) of X.
SIN	Returns the sine in radians.
SQR	Returns the square root.
TAN	Returns the tangent.

STRING FUNCTIONS

ASC	Returns string to ASCII value conversion.
CHR$	Returns ASCII value to string conversion.
CVI, CVS, CVD	Converts string values to numeric values.
EOF	Returns-1 (true) if the end of sequential file is reached.
HEX$	Returns decimal to hexedecimal conversion.
INPUT$	Reads characters from the keyboard.
INSTR	Searches for substring.
LEFT$	Returns leftmost characters.
LEN	Returns length of string.
LOC	Returns the record number just read or written from a GET or PUT statement.
LOF	Returns the length of the file in bytes.

BASIC LANGUAGE SUMMARY

Functions

STRING FUNCTIONS

MID$	Returns a substring of string.
MKI$, MKS$, MKD$	Converts numeric values to string values.
OCT$	Converts decimal to octal.
RIGHT$	Returns right most characters.
SPACE$	Returns string of spaces.
STR$	Returns string representation.
STRING$	Builds string.
USR	Calls Assembly Language Subroutine.
VAL	Returns numerical representation of the string.

SPECIAL FUNCTIONS

CSRLIN	Returns current line position of the cursor.
DATE	Retrieves an integer value representing the current day as defined by DATE$.
FRE	Returns the number of bytes in memory that are not being used by BASIC.
INP	Returns input from port.
LPOS	Returns the position of the print head.
PEEK	Reads a byte from the memory address.
POKE	Puts a specified byte into memory at a specified location.

POINT	Reads the attribute value of a pixel from the screen.
POS	Returns the current cursor position.
SCREEN	Returns the integer value of the specified character.
SPC	Prints blanks on the terminal or the line printer.
TAB	Spaces to a position of the terminal or line printer.
TIME	Retrieves an integer value representing the current second of the day as defined by TIME$.
VARPTR	Returns an address value which can be used to locate where the variable <variable name> is stored in memory.
WIDTH LPRINT	Sets the printed line width for the printer.

VARIABLES

ERR and ERL	Traps an error by returning an error code and line number associated with an error.
INKEY$	Reads one character from the keyboard.

This provides a summary of the various commands statements, functions, and variables found in Z-BASIC. They have been listed here to demonstrate their functional relationship to each other.

In the "Alphabetical Reference Guide", which follows, you will find each command, statement, function and variable, along with the arguments, and details of how to use them in your programs. An argument is a variable upon whose value the value of a function, command or statement depends. The arguments for Z-BASIC commands are found in the format statements in the Briefs that precede the commands, statements, etc.

BASIC LANGUAGE SUMMARY

Color and Graphic Statements

COLOR	Selects foreground and background color for screen display.
CIRCLE	Draws an ellipse with a center and radius as specified by the arguments.
DRAW	Permits the drawing of graphic images on the screen.
GET	Transfers the screen image into an array.
LINE	Permits the drawing of lines boxes and filled boxes on the screen.
PAINT	Fills graphic figures with the specified paint attribute until it reaches the specified border attribute.
PRESET	Turns off a point at a specified location on the screen.
PSET	Turns on a point at a specified location on the screen.
PUT	Transfers image stored in an array onto the screen.
SCREEN	Changes the screen to H-19 graphics mode or reverse video.

CHAPTER 10 ALPHABETICAL REFERENCE GUIDE

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
```

ALPHABETICAL REFERENCE GUIDE

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
```

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
```

ALPHABETICAL REFERENCE GUIDE

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:E000
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 <filespec>,<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.

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,&HC8000
```

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.

ALPHABETICAL REFERENCE GUIDE

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.

ALPHABETICAL REFERENCE GUIDE

CHAIN Statement

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 10.54 and INT, Page 10.82. 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 .4844 and will give a visual circle, assuming a standard monitor screen aspect ratio of 31/64.

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 invisib!e. 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 10.163, for the STOP statement.

COS Function

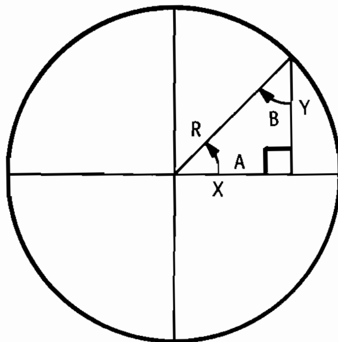
BRIEF

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 10.109, 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 10.144)

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 10.149).

See the examples in the discussion of the READ statement, Page 10.144.

ALPHABETICAL REFERENCE GUIDE

DATE Function

BRIEF

Format: <var> = DATE

Purpose: DATE statement may be used to retrieve the numerical value of the current day of the year as defined by DATE$.

<var> is an integer variable.

Details

The current date, as defined by DATE$, is returned and assigned to the integer variable as the numerical value for that day of the year.

DATE can assume any value from 1 to 366.

If DATE$ = "01-01-82", then DATE will equal 1

If DATE$ = "12-31-82", then DATE will equal 365 (366 for a leap year)

DATE cannot be assigned a value directly. However, the value of DATE changes any time a new assignment is made to DATE$.

EXAMPLE:

```
DATE$="10-28-82"
OK
PRINT DATE$, DATE
10-28-1982      301
OK
```

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>= DATE$, DATE$ 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
```

ALPHABETICAL REFERENCE GUIDE

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.

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. **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
```

ALPHABETICAL REFERENCE GUIDE

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 10.124).

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.

ALPHABETICAL REFERENCE GUIDE

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.

ALPHABETICAL REFERENCE GUIDE

DRAW Statement

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 .
```

EDIT Command

Line number is the program line nurnber 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
```

ALPHABETICAL REFERENCE GUIDE

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.

ALPHABETICAL REFERENCE GUIDE

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
```

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 alphanumeric 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:

```
PRINTFIX(58.75)
 58
Ok

PRINTFIX(-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.

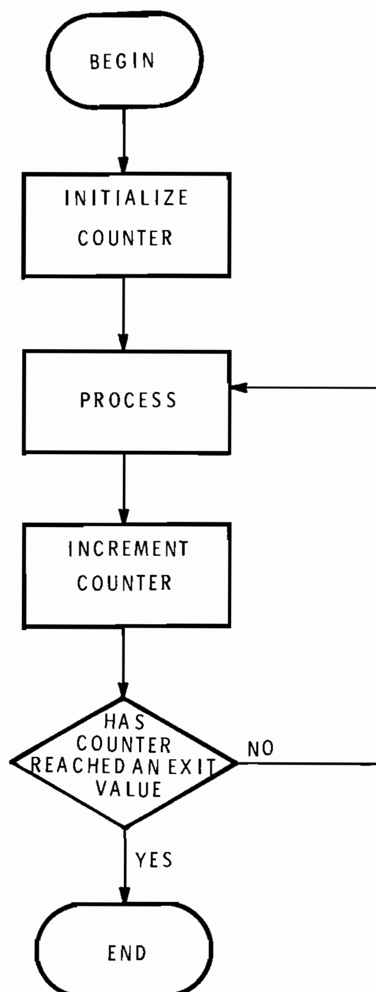
Checks

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.

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.

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 10.56, 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 columns 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 itself 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).

ALPHABETICAL REFERENCE GUIDE

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.

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.

ALPHABETICAL REFERENCE GUIDE

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 10.80.

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 10.125.

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 IS95.13149
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
```

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
```

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.

ALPHABETICAL REFERENCE GUIDE

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 10.54, 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$" Page 10.107 and "RIGHT$," Page 10.152.

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
  .
  .
  .
```

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 10.93, LINE INPUT#.

ALPHABETICAL REFERENCE GUIDE

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?  LINDA JONES   234,4  MEMPHIS
LINDA JONES   234,4   MEMPHIS
Ok
```

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.

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 10.94.

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 80.

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.

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 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
```

ALPHABETICAL REFERENCE GUIDE

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 10.132 — 10.137.

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 10.109.

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.

ALPHABETICAL REFERENCE GUIDE

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$, Page 10.87 and RIGHT$, Page 10.152.

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

ALPHABETICAL REFERENCE GUIDE

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 10.66.

ALPHABETICAL REFERENCE GUIDE

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
```

ON ERROR GOTO Statement

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.

ALPHABETICAL REFERENCE GUIDE

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.

OPEN COM Statement

<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.

ALPHABETICAL REFERENCE GUIDE

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 10.129.

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: `POKEI,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 command to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. See Page 10.127.

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 10.141. 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.

ALPHABETICAL REFERENCE GUIDE

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.

ALPHABETICAL REFERENCE GUIDE

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".
```

PRINT# and PRINT# USING Statements

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
```

ALPHABETICAL REFERENCE GUIDE

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
```

PSET Statement

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.

ALPHABETICAL REFERENCE GUIDE

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 10.149).

READ Statement

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 10.64 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 ASCII value of the character that is located at the specified row and column on the screen.

Details

X is a numeric variable receiving the integer returned.

row is a number between 1 and 25, the row number.

col is a number between 1 and 80, the column number.

z is an optional number between 0 and 255, which, if present and not zero, will cause the function to return the color attributes of the location instead of the ASCII value of the character.

NOTE: Any values entered outside these ranges will result in an `Illegal Function Call` error.

Action:

The integer value of the ASCII character at the specified location is stored in the variable. If the optional parameter <z> is given and not zero, a single byte, containing color attribute information is returned.

Example:

`100 X=SCREEN(10,10)`	If the character at location 10,10 is A, then 65 will be returned.
`110 X=SCREEN(1,1,1)`	Return the color attribute of the character located in the upper left-hand corner of the screen.

For more information, see the discussion on the "SCREEN Function" on pages 7.5 and 7.6.

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:

```
PRINTSIN(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 10.161.

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 10.160.

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.

ALPHABETICAL REFERENCE GUIDE

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 Function

BRIEF

Format: <var> = TIME

Purpose: TIME statement may be used to retrieve the numerical value of the current second of the day as defined by TIME$.

<var> is an integer variable.

Details

The current second of the day, as defined by TIME$, is returned and assigned to the integer variable as the numerical value of that second within one day.

TIME can assume any value from 0 to 86,399.

If TIME$ = "00:00:00", then TIME will equal 0.

If TIME$ = "00:00:59", then TIME will equal 59.

If TIME$ = "00:02:07", then TIME will equal 127.

TIME cannot be assigned a value directly. However, the value of TIME changes any time a new assignment is made to TIME$.

EXAMPLE:

```
TIME$="00:05:10"
OK
PRINT TIME$, TIME
00:05:13      313
```

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.

TIME$ Statement

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
```

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 10.38

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))
```

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:

ALPHABETICAL REFERENCE GUIDE

VARPTR Function

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 1st 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 10.132.

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.

APPENDIX A

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.

APPENDIX A

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$,I,J-I+1)=X$`

APPENDIX B

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.

Field Variables Data

The data that is in field variables when a record is read from a file is set to null values when the file is closed. This is in contrast to MBASIC, where the data in the field variable is preserved when the file is closed.

Clear Screen

The escape sequence that clears the screen in Z-BASIC leaves the cursor at the current position. Whereas in MBASIC the cursor is left in the home position.

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.	(see below)

```
{---------}
{         }
{         }
{   ***   }
{  *****  }
{  *****  }
{  *****  }
{   ***   }
{         }
{         }
{---------}
```

APPENDIX C

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.	

APPENDIX C

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.	

APPENDIX C

ASCII Character Codes and Graphic Symbols

OCT	DEC	HEX	CHAR	KEY	CTRL	DESCRIPTION
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.

SYMBOL

155:
```
{---------}
{         }
{         }
{         }
{         }
{****     }
{****     }
{****     }
{****     }
{****     }
{---------}
```

156:
```
{---------}
{****     }
{****     }
{****     }
{****     }
{****     }
{         }
{         }
{         }
{         }
{---------}
```

157:
```
{---------}
{     ****}
{     ****}
{     ****}
{     ****}
{     ****}
{         }
{         }
{         }
{         }
{---------}
```

160:
```
{---------}
{*********}
{*********}
{*********}
{*********}
{*********}
{         }
{         }
{         }
{         }
{---------}
```

161:
```
{---------}
{     ****}
{     ****}
{     ****}
{     ****}
{     ****}
{     ****}
{     ****}
{     ****}
{     ****}
{---------}
```

162:
```
{---------}
{*********}
{*******  }
{******   }
{*****    }
{****     }
{***      }
{**       }
{*        }
{         }
{---------}
```

163:
```
{---------}
{         }
{         }
{         }
{         }
{*********}
{   **    }
{   **    }
{   **    }
{   **    }
{---------}
```

APPENDIX C

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.	`{--------}`
							`{********}`
							`{********}`
							`{        }`
							`{        }`
							`{        }`
							`{        }`
							`{        }`
							`{        }`
							`{        }`
							`{--------}`

APPENDIX C

ASCII Character Codes and Graphic Symbols

<table>
<tr><th>OCT</th><th>DEC</th><th>HEX</th><th>CHAR</th><th>KEY</th><th>CTRL</th><th>DESCRIPTION</th><th>SYMBOL</th></tr>
<tr><td>173</td><td>123</td><td>7B</td><td>{</td><td>{</td><td>...</td><td>Left brace.</td><td><pre>
{---------}
{   }
{   }
{   }
{   }
{   }
{   }
{   }
{*********}
{*********}
{---------}
</pre></td></tr>
<tr><td>174</td><td>124</td><td>7C</td><td>¦</td><td>¦</td><td>...</td><td>Vertical bar (broken).</td><td><pre>
{---------}
{**   }
{**   }
{**   }
{**   }
{**   }
{**   }
{**   }
{**   }
{**   }
{---------}
</pre></td></tr>
<tr><td>175</td><td>125</td><td>7D</td><td>}</td><td>}</td><td>...</td><td>Right brace.</td><td><pre>
{---------}
{   **}
{   **}
{   **}
{   **}
{   **}
{   **}
{   **}
{   **}
{   **}
{---------}
</pre></td></tr>
<tr><td>176</td><td>126</td><td>7E</td><td>~</td><td>~</td><td>...</td><td>Tilde.</td><td><pre>
{---------}
{   }
{   **** }
{ ****   }
{ ****   }
{   ***   }
{   **   }
{   **   }
{   }
{   }
{---------}
</pre></td></tr>
<tr><td>177</td><td>127</td><td>7F</td><td>DEL</td><td>DELETE</td><td>...</td><td>Delete (rubout).</td><td></td></tr>
<tr><td>200-
377</td><td>128-
255</td><td>80-
FF</td><td></td><td></td><td></td><td>unassigned.</td><td></td></tr>
</table>

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>.

APPENDIX E

Assembly Language Subroutines

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

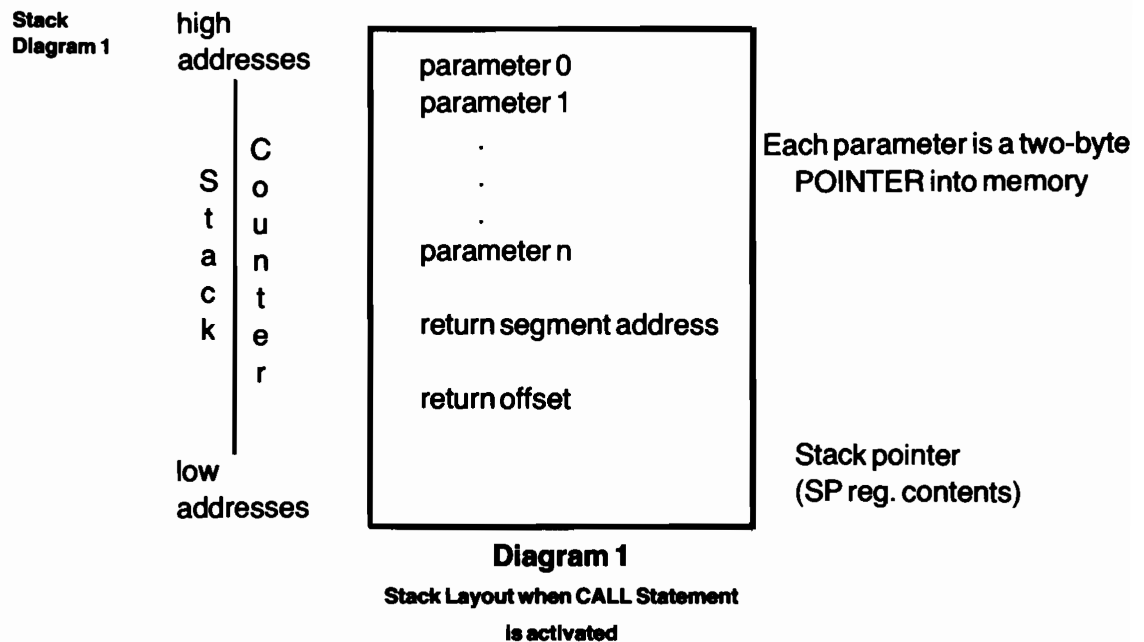


Diagram 1

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).

APPENDIX E

Assembly Language Subroutines

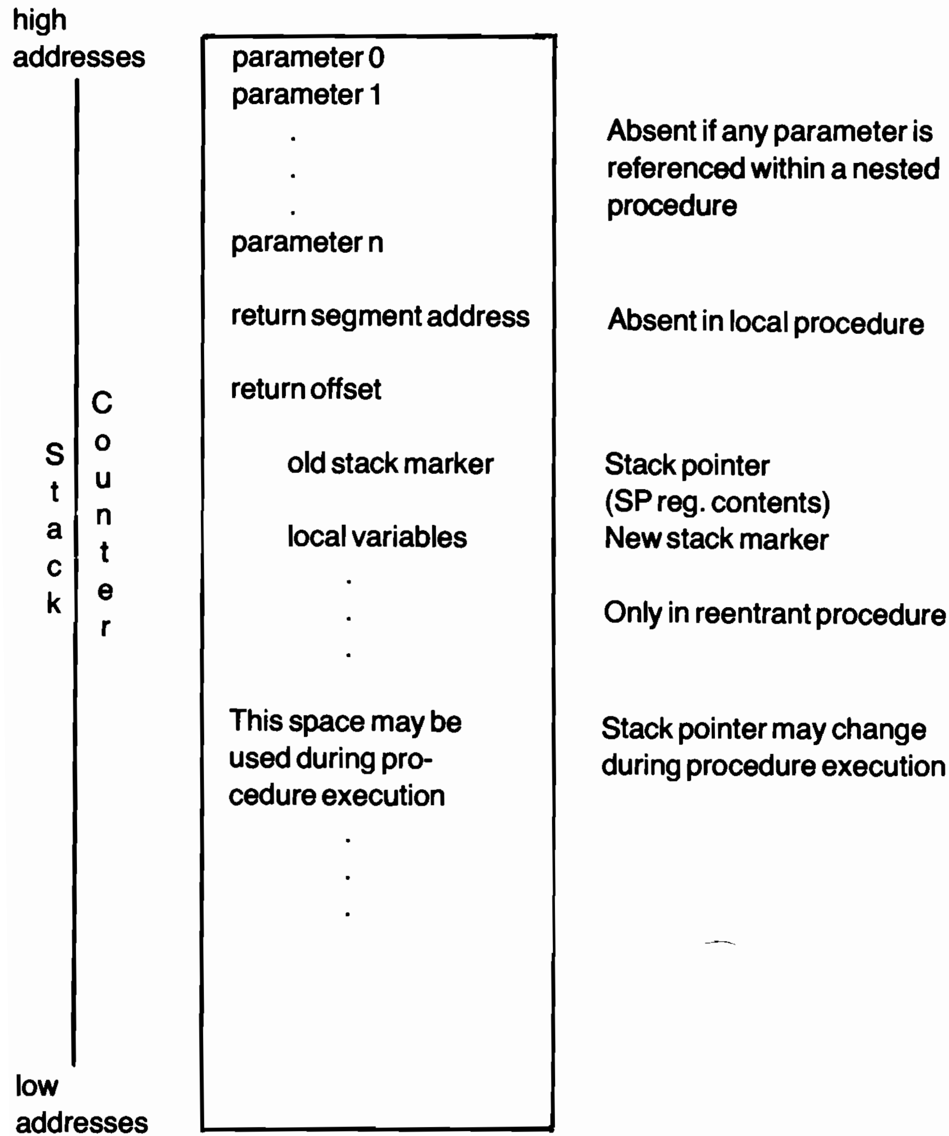


Diagram 2

Stack Layout During Execution of a CALL statement

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, DI, 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.

APPENDIX E

Assembly Language Subroutines

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 version1.5
```

APPENDIX E

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 param1
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
```

APPENDIX E

Assembly Language Subroutines

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$=INPUT(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.

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.

Line No.	Comments
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 (1–16). 1 through 12 are the soft keys F1 through F12 and 13 through 16 are the arrow keys.

We add the following statements:

```
ON <event specifier> GOSUB <line number>
```

APPENDIX F

Communication I/O

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.

APPENDIX F

Communication I/O

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.

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.

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

A

A option, 10.155
ABS function, **10.1**
Absolute address, 7.9,8.16
Absolute value, 10.1
Action verb, 8.24,10.61
Adding Data to a Sequential File, **6.14**
Addition, 5.22
Advanced graphics, 8.1,8.33
Algebraic expressions, 5.23
ALL option, 10.13,10.23
AND, 5.32-**5.35**,5.37-5.45,8.25,10.61
Angles, 8.9,8.10
Angle brackets, 2.18
Angle command, 8.17,8.20,10.42
Angles of a Circle, 8.9,8.10,10.17
Angle parameters, 8.9,10.17
Animation, 8.14,8.22,8.23,**8.25**10.62
Append a P, 4.9,10.155
Append an A, 4.9,10.155
Application programs, **1.10**
Application
  definition of, **1.2**
Argument, 5.46,9.11
Arithmetic Functions, **5.46**
Arithmetic operations, 5.29
Arithmetic operators, **5.19**-5.24
Array, 5.1,**5.12**-5.18,8.14,8.22,
    10.61,10.123
Array Declarator, **5.12**-5.13
Array size formula, 8.23,8.24
Array storage allocation, **5.14**
Array Subscript, **5.13**,10.40
Array variables, 5.21,10.23,10.40
Array
  one-dimensional vertical, 5.12
ASC, 5.57,6.7,**10.2**
ASCII, 4.9,5.31,**5.57**,6.16,6.26,
    7.1,**10.2**,10.155,10.165
Aspect ratio, **8.10**,8.11,10.17
Assembly language, 1.4,**1.5**,10.10,10.11
Assembly language programs, 10.10
Assembly language routines, 10.24
Assigned values, 5.1
Assignment and Allocation Statements, **9.3**
Asynchronous communication, 2.13,10.121
ATN function, 10.3
Attribute value, 8.1-8.13,10.17,10.126
AUTO command, 4.5,10.4

B

BACK SPACE, 2.17,3.6,3.7,3.10
Background color, 8.1,8.3,10.21,10.22
Backslash (\), 5.19,5.20,5.22
Bar graph, 8.8
Base-two, 5.39
BASIC Command Mode, 2.4
BASIC statement, 2.10,2.11
Batch mode, 2.2
BEEP statement, 10.5
Bell, 10.5,10.15
Binary code, **1.4**,5.57
Binary file, 4.1
Binary format, 4.9
Binary notation, 5.39
Binary operator, 5.21
Bit, **5.32**
Bit manipulation, 5.38
Bit patterns, 5.38,5.40,5.41,5.57
BLOAD command, **10.6**,10.7,10.8
Boot up, 4.1
Border, 7.2,8.6
Border attribute, 8.12
Boundaries, 8.12,8.22
Box option, 8.6
Braces, 2.18
Branch commands, 1.6
Brief
  definition of, **1.1**

INDEX

BSAVE, 10.6,**10.8**
BU%, 6.25
Buffer, 6.6,6.17,6.20,6.21
Bug, 4.7
Bytes, 8.24
Bytes free number, 4.1

C
Calculator, 2.5
CALL statement, 10.10,10.11,10.24
Capital letters, 2.18,5.31
CAPS, 2.18
Carriage return, 3.3,3.4
CDBL function, 5.47,10.12
Changing a Z–BASIC Program, **3.4**
CHAIN Statement, **10.13**,10.23
Character Image display program, 8.27
Character set, **2.7**
Checkpoint
  definition of, 1.2
CHR$ function, 5.57,7.5,10.2,**10.15**
CHR$(34), 10.139
CINT function, **10.16**
CIRCLE Statement, 8.5,8.9-8.11,10.17
CLEAR Command, **10.18**
CLOSE statement, 6.3,6.4,6.8,10.19
CLS statement, **10.20**
Colon, 3.1
COLOR statement, 8.1-8.13,8.19,9.12,**10.21**,10.22
Comma, 2.19,2.20,6.13,10.132
Command level, 2.4
Command line options
  <highest memory location>, 2.1,4.2,4.3
Command line with options, 2.1
Commas, 5.55,6.13,10.132
Comments, 2.10,2.11
COMMON statement, 10.13,10.14,**10.23**
Compatibility, 7.3
Compiler, 1.6,1.7,10.14,10.47,10.154
Compressed binary, 4.9,6.2,10.155
Computer languages, **1.4**
Conditional Execution Statements, 9.5
Conditional Branching, 10.65
Cone, 8.11
Conjuction operator, 5.35
Constants, 5.1,5.48-5.50
CONT command, 10.25
  4.8,10.163
Contents of an array, 8.26
Content Organization, **1.2**
Control Characters, 2.17
Control Statements, **5.4**
Control-C, 2.17
Control-G, 2.17
Control-H, 2.17
Control-I, 2.17
Control-J, 2.17
Control-S, 2.17
Control-U, 2.17
Conversion functions, 6.27,10.30,10.109
Converting a numeric constant, 3.51-3.53,5.51-5.53
COS function, **10.26**
Creating a Random File, **6.18**
Creating a Sequential Data File, 6.7
CSNG function, 5.47,10.28
CSRLIN function, 7.12,**7.14**,10.29
CTRL-C, 3.4,3.7,3.10,8.7,10.4,10.94
CTRL-E, 3.7,3.9
CTRL-F, 3.7,3.8
CTRL-G, 2.17
CTRL-L, 3.6-3.8
CTRL-N, 3.7,3.9
CTRL-U, 3.6,3.7,3.10
CTRL-W, 3.7,3.10
Cursor, 3.1-3.4,3.6,4.7,10.29
Cursor movement, 3.6,3.7
CVD, 6.27,10.30
CVI, 6.27,10.30

CVS, 6.27,10.30

D

DATA statements, 8.8,10.31,10.144,10.149
Data type definition statements, 9.3
DATE Function, 10.32
DATE$ statement, 10.33
Debugging, 1.8,4.7,4.8,10.173
Decimal point, 10.135
Declaring Variable Types, **3.7**
Default, 2.12
Default aspect ratio, 8.10,10.17
Default attributes, 8.5,8.9
Default drive, 4.1,4.9
Default extension, 2.2,2.15,4.9
DEFDBL statement, 9.3,10.36
DEF statement, 9.3
DEF type statement, 9.3,10.36
DEF SEG statement, 10.6,10.10,**10.37**
DEF USR statement, 10.38
DEF USR0 statements, **10.38**
DEFINT, 9.3,10.36
DEFSTR, 9.3,10.36
DELete Key, 3.6,3.7,3.9
Deleting characters, 3.6,3.9
DELETE option, 10.14
DELETE command, **10.39**
Delimiters, 2.19,2.20
DEMO I, 8.30,8.31
DEMO II, 8.32,8.33
Details
  definition of, **1.1**
Device, 2.12,2.13
Device name, **2.12**,2.13
Device specification, 2.13,4.9
DIM statements, **5.12**-5.15,10.24,10.40
Dimensions, 5.12,5.17,8.22
Direct Mode, 2.4,**2.5**,3.2-3.5
Directory pointer, 6.7
Disjunction operator, 5.35,5.37,5.42
Disk directory, 6.7
Disk I/O, 6.17
Disk sector, 6.6
Display format, 7.1
Displaying graphic images, 10.6,10.8
Division, **5.21**,5.22
Division by zero, **5.22**
Dollar sign, **5.8**,5.54,10.136
Double asterisk, 10.136
Double dollar sign, 10.136
Double-precision, 5.1,5.48-5.53,8.23,9.3,
  10.28,10.30,10.109
Double-precision constant, 5.49
Double-precision numbers, 5.48
Double-precision variables, **5.10**,5.51
Double quotation marks, 5.48
Draw statement, 8.14-8.21,8.23,10.41,10.42
Drive number, 6.7
Drive specification, 4.10
Duplicate definition error, 5.13

E

EDIT command, 3.1,3.2,10.43
Edit Mode, 4.7,10.43
Editing Z-BASIC, 3.1-3.11
Element, 5.1,5.12
Ellipse, 8.9-8.11
ELSE clause, 10.67,10.69,10.70
END statements, 10.45
End-of-data marker, 6.5
EOF function, 10.46
EOF pointer, 6.9
Equivalence, 5.37
Equivalence table, 5.40
EQV operator, 5.37,5.44
ERASE statement, 10.47
ERASEing, 5.13
ERR and ERL variables, 9.11,10.48
ERROR statement, 10.49, 10.50

INDEX

Error trapping, 4.7,4.13,6.15, 10.114
Event Trapping, 10.151
  See also Appendix F
Exceptions to Naming Variables, 5.2
Exclamation point, 5.1
Exclusive OR operator, 5.35
Executable statements, **2.11**
Execution error, 5.26
EXP function, 10.51
Exponentiation, 5.20
Exponentiation Functions, 5.47
Expressions, 5.27
Extension, 2.12,2.14,6.7
Extension .BAS, 2.15,4.6,4.9

F
Field, 2.20,6.17,6.22
Field buffer, 10.176
FIELD statement, 6.20,6.21,6.24,6.25,**10.52**
FIELD string, 6.21
Field variables, 6.25
Field-structured, 6.17,6.22
File, **2.12**
File buffers, 6.6,6.7
File Control Block, 10.177
File Management Statements, **6.3**
File Manipulation Commands, **6.1**
File naming conventions, 2.15
File structure, 6.5
Filename, 2.14,2.15,4.6,4.9,
  4.10,6.1
FILES command, 6.1,10.53
Filespec, **2.12**
Filling a graphic figure, 8.12,8.13,10.126
FIX function, 10.54
Fixed Point, 5.49
Flag, 5.38
Flicker, 8.25
Floating point, 5.49,5.50
Floating Point Division, 5.22
FOR...NEXT statements, 10.55-10.58
Foreground color, 7.10,8.1,8.3,8.4
  8.12,10.21
Formatting printed output, 10.138
Four carats, **10.137**
FRE function, 10.59
Free memory, 4.2
Full Screen Editor, 3.1-3.11
  Deleting text, 3.6
  Inserting text, 3.6
  Key assignments, 3.6
Functions, 5.46-5.47,9.8-9.10

G
GET statement, 6.17,6.25,10.60
GET/PUT, statement, 8.14,8.22-8.29,
  10.61-10.63
Getting Records Out of the File, 6.24
GOSUB...RETURN statement, 10.64
GOTO statement, 4.6,4.8,**10.65**
Graphic Transfer, 8.22
Graphic Statement, 8.1,8.12,8.13
Graphic Symbols, 7.4
Graphics Macro Language, 8.14,10.41

H
H-19 Graphics mode, 7.1,7.4,10.157
Hex constants, **3.49**
HEX$ function, 10.66
High-level language, 1.5
Highest memory location, 2.3
Highlighting, 7.3
Holes, 8.12
HOME key, 3.6-3.8
Horizontal, addressable points, 7.1,7.2

INDEX

I
I/O statements, 10.117
IF...THEN statements, 10.67-10.71
I/O buffers, 6.5-6.7
I/O devices, 6.6
I/O statements, 6.3,9.6
Illegal Function Call, 5.13
Image storage procedure, 8.23
Image transfer, 10.61
Immediate mode, 2.4
IMP operator, 5.36,5.43
Inaccuracies, 5.30
Indirect Mode, 4.4
Initialization, 2.2
INKEY$ variable, 10.72
INP function, 10.73
Input buffers, 6.6
INPUT statement, 10.74-10.78
INPUT# statement, 6.10-6.13,10.79
INPUT$ function, 10.80
Inputting Z-BASIC Programs, 3.2
Insert Mode, 3.6,3.9
Inserting text, 3.6
Integer constants, 5.49
INSTR function, 10.81
INT function, 8.22,9.21,10.82
Integer, 8.23,10.16,10.54,10.82
Integer Array, 8.26
Integer Division, 5.19,5.20,5.22
Integer value, 5.12
Integer variables, 5.9
Integers, 5.1,5.9,5.48,5.51,
10.16,10.82
Internal Representation, 5.38,5.40,5.41,6.26
Interpreter, 1.4,1.5,**1.6**,1.8,1.9,4.1,4.3,
4.7,4.10,5.23,5.25,6.5,6.25
Intrinsic function, 5.46
I/O devices, 10.121

K
KEY statement, 10.83-10.85
Key values, 3.7
Key-pad keys, 3.1
KILL command, 6.1,10.86

L
LEFT$ function, 3.56,**10.87**
Left-justified, 6.22,6.23,10.105
LEN, 3.54,5.56,10,88,10.107
LET, **2.5**,4.4,10.89
LET statement, 10.89
Line folding, 3.9
LINE INPUT statement, 10.92
LINE INPUT#, 6.15,10.93
Line number, 2.6,**2.10**,2.11,3.1,3.2,
4.4,4.5,4.6
Line-feed, 3.3
LINE statement, 7.2,8.5-8.8,10.90-10.91
LIST command, 3.2,10.94
Listing a BASIC Program to a Line Printer, 4.11
Literal quotes, 6.13
LLIST command, 4.11,10.96
LOAD command, **6.1**,10.97
Loading a BASIC Program, 6.1
Loading the BASIC Interpreter, 4.2
LOC function, 6.3,6.4,6.17,**10.98**
LOCATE statement, 7.12,7.13,10.99,10.100
LOF function, 6.3,6.4,10.101
LOG function, **10.102**
Logic error, 4.7
Logical line, 3.3-3.5
Logical operators, 5.32-5.45
Lower case, 3.30
Lower case letters, 5.31,7.4
LPOS function, 10.103,10.181
LPRINT statement, 10.104
LSET statement, 6.18,6.21,6.22,10.105

INDEX

M

Machine independent, **1.5**
Machine level, 5.37-5.38
Mathematical functions, 9.8
Matrix Manipulation, 5.16
Memory, 1.5,4.2,4.3,4.6,6.2,6.6
Memory space, 5.8
Memory space requirement, 5.51-5.53
MERGE command, **6.2**,10.106,10.155
MERGE option, 10.14
Microprocessor, 1.6
MID$, function, 10.107
MID$, statement, 5.56,10.108
Minus sign, 10.135
MKD$ function, 10.109
MKI$ function, 10.109
MKS$ function, 10.109
Modulo Arithmetic, 5.19
Modulus division, 5.20,5.22
Movement Commands, 8.15-8.19,10.43
Multi-Dimensional Arrays, 5.14
Multiplication, 5.21

N

N spaces/, 10.134
Name command, 6.2,10.110
Negation, 5.21,5.27
Nested FOR NEXT statement, 10.57,10.58
Nesting of subroutines, 10.64
NEW command, 10.19,**10.111**,10.173
Non-executable statements, **2.11**
NON-I/O Statements, 9.5
Non-local RETURN, 10.151
NOT, 5.32-5.34,5.37
Notation, 2.18
Null statement, 10.112
Number sign, 10.135
Numeric comparison, 5.30
Numeric constants, 5.47-5.49
Numeric expressions, 5.19,5.20,5.24
Numeric Fields, 10.135
Numeric Functional Operators, 5.47
Numeric value, 5.38
Numeric variables, 5.1,5.5,5.7,5.8,5.52

O

O mode, 6.14
OCT$ function, 10.113
Octal constants, 5.49
Offset, 10.6-10.8
OK, **2.1**,2.4
ON ERROR GOTO statement, 10.114
ON...GOSUB statement, 10.116
OPEN COM statement, 10.121-10.123
OPEN “O” statement, 6.19
OPEN “R”, 6.19
OPEN statement, Z–BASIC, **6.4**,6.7,6.17,**10.117**
OPEN statement, Z–BASIC, 10.117-10.120
Opening a File for Random Access, 5.19
Operands, 5.19,5.21,5.22,5.41
Operating system, 6.6
Operators, 5.19-5.47
Optimization, **1.7**
OPTION BASE Statement, 5.13-5.15,**10.123**
Options, **2.2**
OR operator, 5.35,5.38,5.41,5.43
Order of precedence, 5.20,5.37
OUT statement, 10.124
Output buffers, 6.6,6.7,6.10
Overflow, 5.22,5.52
Overlay, 10.106

P

P options, 4.9,6.3,10.155
Packed binary format, 2.16
Pad out, 6.21

INDEX

PAINT statement, 8.5,8.12,8.13,10.126
Painting jagged edges, 8.13
Parentheses, **5.19**,**5.23**,5.24,5.25,5.37,5.38
Parity, 10.121
PEEK function, 10.127
Percent sign, 10.137
Period, 3.2,10.43
Peripheral device, 2.12
Physical Organization, **1.1**
Physical record, 6.6,6.7
PI, 5.5,8.9,8.10
Pixels, 7.1,7.7,8.22,8.23,
    10.63,10.128
Plotting Coordinates, 7.1-7.14,8.30
Plus sign, 10.135
Pointers, 6.6,6.7,6.21
POINT function, 7.7,7.8,10.128
POKE function, 10.126,10.129
POS function, 7.12,7.14,10.29,10.130
Precedence, 5.20,5.37
Prefix commands, 8.19-8.21
PRESET statement, 7.7,7.11,8.13,8.24,10.131
PRINT #, 6.12-6.14,10.138
PRINT CHR$(7), 10.5
Print positions, 10.132,10.134,10.135
PRINT statement, 2.6,2.19,5.25,5.55,6.8,
    10.132-10.133
PRINT USING statement, 10.134-10.137
Print zones, 2.20
PRINT# statement, 6.8-6.12
PRINT# and PRINT# USING, 10.138-10.139
Printing/formatting techniques, 2.19
Printed numbers, 10.135
Problem oriented, **1.5**
Program development process, 1.8,1.9
Program line, 3.3-3.4,4.4
Program line format, 4.4
Programming language, **1.4**,1.5
Prompt string, 10.74-10.75
Protect option, 4.9,6.3,10.155
Protected Files, 6.3
PSET statement, 7.7-7.10,8.7,8.24
    10.62,10.140-10.141
Punctuation, 2.18,5.31
PUT and GET statements, 8.22,10.61-10.63
PUT Statement, 6.17,6.22,6.25,10.142

Q

Question mark, 5.25,10.53,10.174
Quotation marks, 5.54,6.13,10.139

R

R option, 4.10,6.1,6.2,10.97
Radius, 8.9,8.10,10.17,10.26
Random access, 6.4,6.16
Random access - buffer, 6.17
Random access files, 6.16-6.29
Random file buffer, 6.17
RANDOMIZE statement, 10.143
READ statement, 10.31,10.144
Recognized characters, **2.14**
Record, 6.16,6.17,6.25
Record number, 6.4
Relational expressions, 5.37
Relational Operators, 5.27-5.31
Relative form, 8.7
REM statements, 2.10,2.11,10.146
REMARK statement, 2.10,2.11,10.146
    (see also comment)
RENUM command, 10.147
RESET command, 6.2,10.148
RESTORE statement, 10.31,10.144,10.149
RESUME statement, 10.150
RETURN command, 10.64,10.92,10.151
RETURN key, 3.4,4.1,4.4
Reverse video, 7.3,10.157
RIGHT$ function, 5.56,10.152
Right-justified, 6.22,10.105

INDEX

RND function, 10.153
Rotating figures, 8.18
Row, 7.12,10.29
RSET statement, 6.22,10.105
RS-232 communications, 10.118,10.121,App.F
RUN command, 2.6,**2.11**,4.6,4.10,6.2,10.154
Running a BASIC Program, 4.6

S
SAVE command, 6.2,6.3,10.154,10.155
Saving a BASIC Program, 4.9
Scalar Multiplication, 5.17
Scale factor, 8.20,10.42
Screen display format, 7.1
SCREEN function, 7.1,7.2,7.5,10.156
SCREEN statement, 7.1,7.3,7.4,10.157
Sector, 6.6
Semicolon, 2.19,2.20,5.55,6.12,6.13,
10.132,10.133,10.134
Semicolon terminator, 6.12
Sequence of execution statements, 9.4
Sequential buffers, 6.7,6.17
Sequential data files, 6.5-6.15
Sequential input, 10.117
Sequential I/O, 6.17
Set attribute, 8.17
SGN function, 10.158
SIN function, 10.159
Sign-on, 2.1,4.2
Single-precision, 5.1,5.10,5.48
Single-precision constant, 5.48-5.49
Single-precision number, 8.23,10.30,10.109
Single-precision variables, 5.10
Slash (/), 5.22
SPACE$ function, 10.160
SPC function, 10.161
Space Requirements, 5.51,5.52
Special Functions, 9.10
SQR function, 5.46,10.162
Square brackets, 2.18,4.8
Standard Math Functions, 5.47
Starting Z-BASIC, 2.1
Statement, **2.10**,2.11
STEP, offset, 7.9,7.11,8.7
Stickman, 8.29
STOP statement, 4.8,10.163
Storage format, 8.22,10.63
Storage and Retrieval of Numeric Data, 6.26
STR$ function, 5.54-5.57,6.26,10.164
String arrays, 5.15
String comparisons, 5.31
String constants, 5.54
String expressions, 5.30,5.54,8.14
String Fields, 10.134
STRING$ function, 10.165
String variables, **5.8**,5.54,5.55,5.56,
6.26,10.72,10.92,10.93
Strings, 3.54
Structuring the Random Buffer into Field, 6.20
Subroutine, 10.64
Subscript, 5.12-5.15
Subscript Out of Range, 5.13
Substring, 5.56,8.21
Subtraction operation, 5.21
Subtractions, 5.22
Superimpose the image, 8.25,10.62
SWAP statement, 10.166
Symbols, 5.3,5.31
Syntax, 2.18
Syntax diagrams, 2.18
Syntax errors, 3.4,4.7,5.25
Syntax notation, 2.18
SYSTEM statement, 10.167

INDEX

T
TAB, 3.6,3.9
TAB function, 10.168
TAB key, 3.9
Tabular data, 2.19,2.20
TAN function, 10.169
Terminator, **6.11**,6.12,6.13
TIME Function, 10.170
TIME$ statement, 10.171-10.172
The Program Development Process, 1.8
Trace flag, 4.8
Translation, 1.6
Transposition of a Matrix, 5.17
Trigonometric Functions, 3.47
TROFF statement, 4.8,10.173
TRON statement, 4.8,10.173
Truncate, 2.14,6.22
Truncation, 2.14
Truth table, 5.32-5.33,5.41,5.42
Truth values, 5.38
Two's complement, 5.40,5.45

U
Unary minus, 5.21
Unary operator, 5.21
Underscore, 10.137
Unquoted strings, 6.13
Updating Sequential Files, 6.14
Using the Z–BASIC manual, 1.3
USR function, 10.174

V
VAL, 5.56
VAL function, 5.56,6.26,10.175
Valid Colors, 8.1,8.3,10.21
Variable name, 5.1,5.8
Variables, 2.5,2.6,3.4,4.8,5.1-5.18
  10.13,10.23
VARPTR function, 10.176-10.178
Vertical addressable points, 7.1
Vertical Arrays, 5.14
Video board, 8.1
Video RAM chips, 8.1
Video Resolution, 7.1
Video screen, 7.1

W
WAIT statement, 10.179
WEND statement, 10.180
WHILE statement, 10.180
WIDTH statement, 10.181
WRITE statement, 10.182
WRITE# statement, 10.183
Writing a BASIC Program, 4.5

X
X axis, 7.1,7.2
XOR, 5.35,5.42,8.25,10.62

Y
Y axis, 7.1,7.2

Z
Z-100, 2.12,2.20
Z-100 All in One monitor, 7.1
Z-100 keyboard, 3.10-3.11
Z-BASIC command line, 2.1
Z-BASIC graphic capabilities, 7.1
Z-BASIC sign on, 2.1
Z-BASIC OPEN statement, 10.117
Z-BASIC summary program, 8.30-8.33
Z-DOS, 4.1,6.6,6.7
Z-DOS AUTOEXEC. BAT, 2.2
Z-DOS filename conventions, 2.15

HEATH | ZENITH data systems

IMPORTANT NOTICE

Dear Customer,

The following unique features of Z-BASIC version 1.00 need to be noted. The following may not necessarily be the same or true in future releases of Z-BASIC.

Invalid Device Names

There may be occasions when a program attempts to access invalid drives such as those with names above drive D (e.g., E, F, G, and so on). Z-BASIC reads this invalid drive name and, instead of generating an error message, accesses the last legal drive that the program or the operator used. It is currently up to the user to ensure that his or her Z-BASIC program accesses only those drives (e.g., A, B...) that are available in their system configuration.

Filename References

Z-BASIC allows a wide range of allowable filenames. Since this feature may not be supported in future releases of Z-BASIC, it is recommended that all file references follow present Z-DOS conventions as defined in your Z-DOS manual.

Color and Optimized Scrolling

The Z-100 Desktop Computer optimizes the scrolling speed of the screen when color is not being used in the system. The computer must be told when Z-BASIC will be working with color on the screen so that it can use the proper scrolling method. Otherwise, the optimized screen scrolling action will cause the color in the display to be lost under certain circumstances.

To make sure that your programs are going to operate correctly, place the following line of code near the beginning of each affected program:

10 CLS:COLOR 1,0:PRINT " ":COLOR 7,0:LOCATE 1,1:PRINT " ":LOCATE 1,1

Programming Note

The 25th line of the display may be assessed while in Z-BASIC. The preferred method to clear (and re-enable) the 25th line is **PRINT CHR$(27);"y1";CHR$(27);"x1";**

Page 1 of 1
MS-463-1
591-3965

Thank you,

Zenith Data Systems