MICROSOFT

# Z-Basic (Z-DOS™)

Volume I

593-0040-01
CONSISTS OF

MANUAL
595-2825-01
FLYSHEET
597-2747-01

Printed in the
United States of America

ZENITH data systems

HEATH

**NOTICE**

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

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

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

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

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

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



HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

ZENITH DATA SYSTEMS CORPORATION
ST. JOSEPH, MICHIGAN 49085

MICROSOFT

# Z-Basic
# (Z-DOS™)

593-0040
CONSISTS OF

MANUAL
595-2825
FLYSHEET
597-2747

Printed in the
United States of America

ZENITH data systems

HEATH

# Volume 1

Preface ..... X
Major Features of Z-Basic ..... XI

## Part I Introduction

**General Information**

**Chapter 1** ..... 1.1
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

# TABLE OF CONTENTS

## Volume 1

## Part II BASIC Overview

### Program Entry

**Chapter 2** ..... 2.1
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

### Editing Z-BASIC Programs

**Chapter 3** ..... 3.1
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

# TABLE OF CONTENTS

## Volume 1

**Chapter 4** ..... 4.1
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

**Programming in Z-BASIC**

**Chapter 5** ..... 5.1
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

**Arithmetic and String Operators**

# TABLE OF CONTENTS

## Volume 1


**File Handling**

**Chapter 6** ..... 6.1
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

**Plotting Coordinates**

**Chapter 7** ..... 7.1
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

**Advanced Color Graphics**

**Chapter 8** ..... 8.1
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

Volume

# Basic Language Summary

**Chapter 9** ..... 9.1
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

# Volume 2

## Part III Reference Guide

**Alphabetical Reference Guide**

**Chapter 10** ..... 10.1

## Part IV Appendices, Glossary and Index

**Appendices**

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
APPENDIX G: Glossary ..... G.1
Bibliography ..... H.1
Index ..... X.1

# 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

# 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.

## 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.

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**

*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.

**Application**

*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.

## 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.

# 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.

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<br>MOV A,B<br>ADD C<br>MOV B,A<br>POP PSW<br>CMP B<br>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.

## 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.

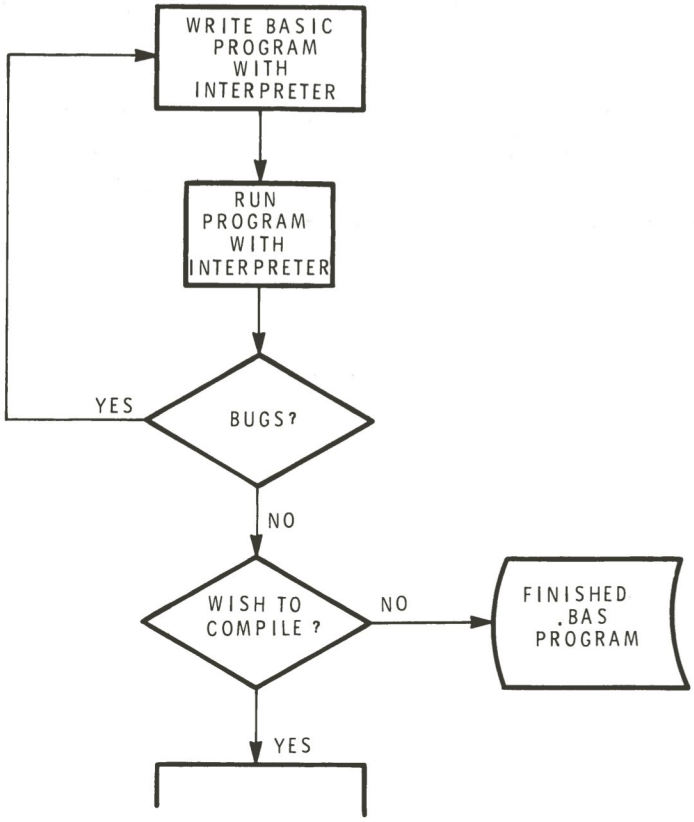



**Figure 1.1**
**The Program Development Process**

## 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.

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";**



Thank you,

Zenith Data Systems

# CHAPTER 2 PROGRAM ENTRY

# 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 from 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. |

# 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 10.87) 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.

# 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 |

## 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 | TO | |
| CVI | GET | LSET | RANDOMIZE | TROFF | |
| CVS | GOSUB | MERGE | READ | TRON | |
| DATA | GOTO | MID$ | REM | 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 |

**Table 2.1**
**Input and Output Devices**

## 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.

**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.

# 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. |

# 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 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.

**Line number**

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

| | |
|---|---|
| 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** |

**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.

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

**CTRL-Z** Erases to the end of the page.

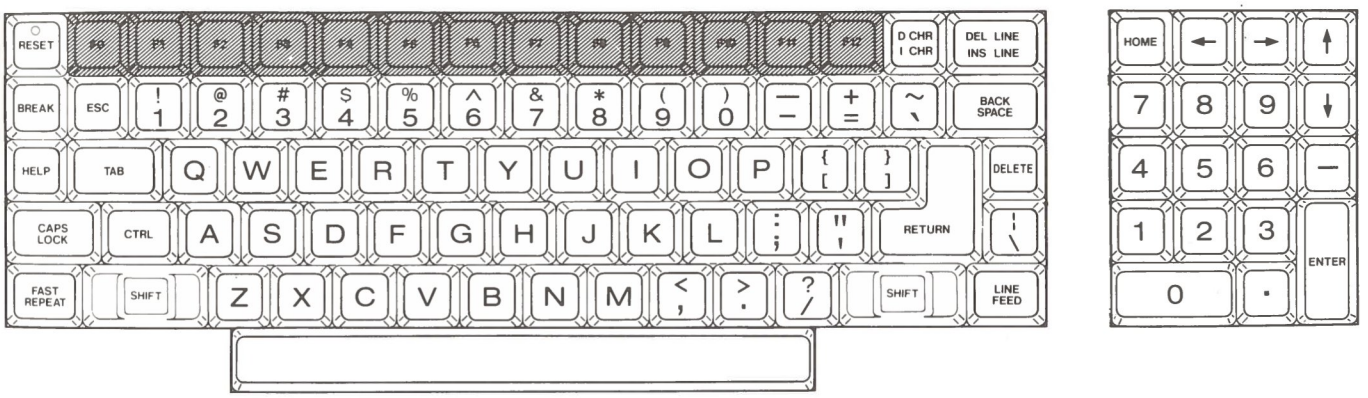



**FIGURE 3.1**
**Function Keys**

# EDITING Z-BASIC PROGRAMS

## The Full Screen Editor — Key Assignments

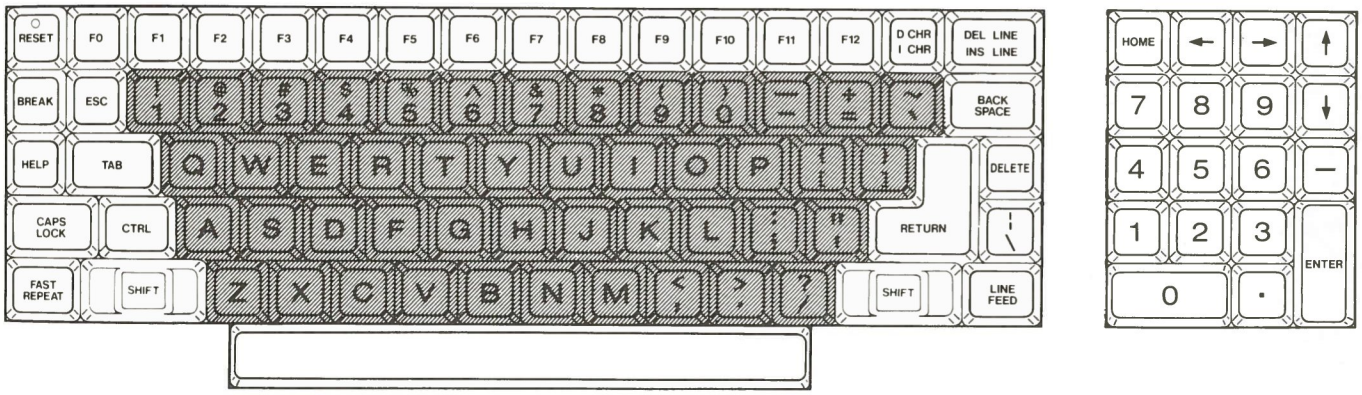



**FIGURE 3.2**
**Alphanumeric Keys**

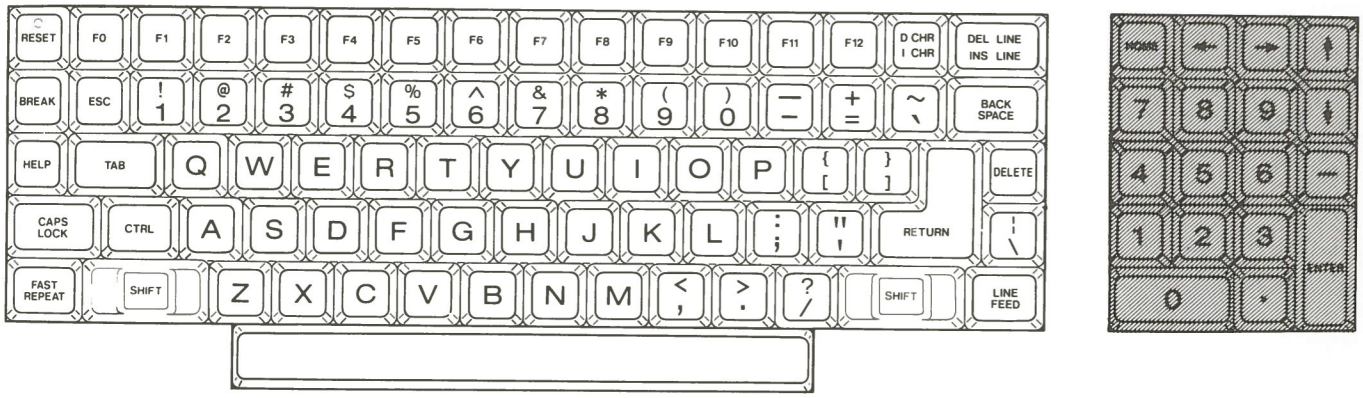



**FIGURE 3.3**
**Keypad**

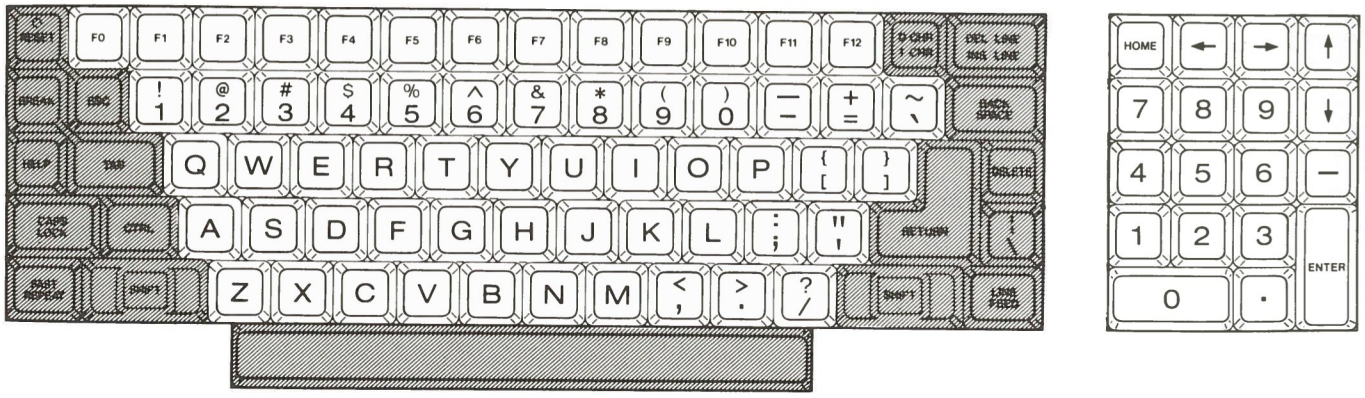



**FIGURE 3.4**
**Special Keys**

# 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.

**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) | \| (space) | |

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.

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.

# 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.

# 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.

# 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.

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.

**Repetition of a Variable in Several Locations with the same Value**

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

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$).

**Passing Values**

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

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"
```

**Declaring Numeric Variable Precision**

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.

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.

# ARITHMETIC AND STRING OPERATORS

## Variables

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

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

**Declaring Single Precision Variables**

```
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

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

**Declaring Double Precision Variables**

Example:

```
DEBIT# = 91283764518.28
```
would cause the value of DEBIT# to be 91283764518.28

```
WORTH# = 998877665544332211.998877665544332211
```
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 $3 \times 3$ 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

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.

**Matrix Input Subroutines**

```
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 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.

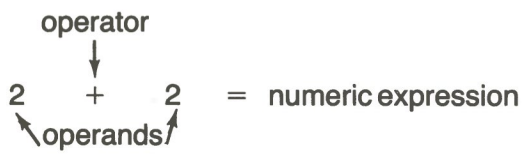
Rules of precedence determine the order in which operators are evaluated.

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**

*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.

| 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 evalulated 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

| | | |
|---|---|---|
| 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).

## Arithmetic Operators and Expressions

**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**

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.

```
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.

# ARITHMETIC AND STRING OPERATORS

## Arithmetic Operators and Expressions

### 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.

# ARITHMETIC AND STRING OPERATORS

## Arithmetic Operators and Expressions

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**

# ARITHMETIC AND STRING OPERATORS

## Arithmetic Operators and Expressions

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**

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 | | Precipitation | |
|---|---|---|---|
| HOT | -1 | NONE | -1 |
| WARM | -1 | RAIN | -1 |
| COOL | 0 | HAIL | 0 |
| FRIGID | 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.

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.

**XOR**

| X | Y | X XOR Y |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

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

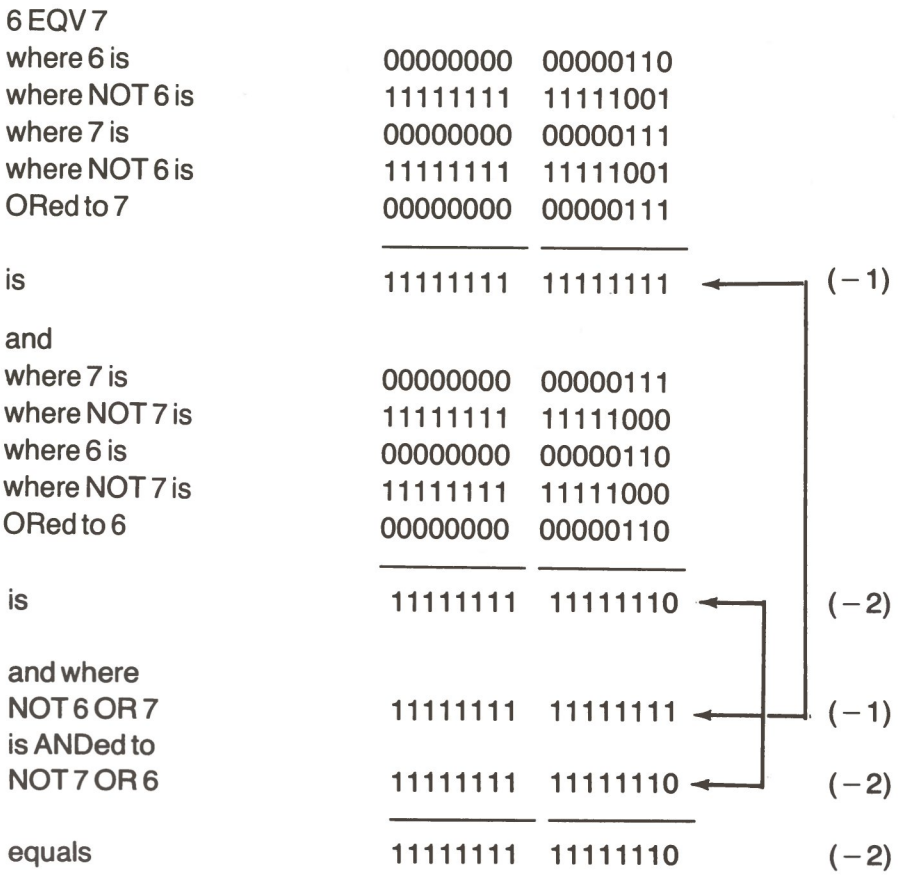
| | | |
|---|---|---|
| where 6 is: | 00000000 00000110 | |
| where NOT 6 is: | 11111111 11111001 | |
| where 7 is: | 00000000 00000111 | |
| where NOT 6 is: | 11111111 11111001 | |
| ORed to 7: | 00000000 00000111 | |
| equals | 11111111 11111111 | (−1) |

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

# ARITHMETIC AND STRING OPERATORS

## Arithmetic Operators and Expressions

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). **EQV**




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 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

```
                    SQR              (4)
                Function name      Argument

        4---------->  SQR(X)  ---------->2
    argument          function          result
```

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**

# 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.

# 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
```

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
PRINT A$;B$;C$
PRINT A;A$;B;B$;C;C$
```

RESULT:

```
 1  2  3
ONETWOTHREE
 1 ONE  2 TWO  3 THREE
```

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

```
PRINT A,B,C,
PRINT A$,B$,C$
```

RESULT:

```
 1          2          3
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$)*VAL(B$)
```

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"
```

**MERGE**

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.

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

**NAME**

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.

```
RESET
```

**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.

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

**RUN**

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.

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

**SAVE**

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.)

## 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**

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).

# 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.

| | | |
|---|---|---|
| 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$` |
| 3. | To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode. | `CLOSE #1`<br>`OPEN "I",#1,"TEST.ASC"` |
| 4. | Use the INPUT# statement to read data from the sequential file into the program. | `INPUT#1,X$,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

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:

**INPUT #**

```
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.

**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.

# 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.

**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**

## 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$=X$`<br>`LSET A$=MKS$(AMT)`<br>`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.

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
```

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.

```
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 639 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 224 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.

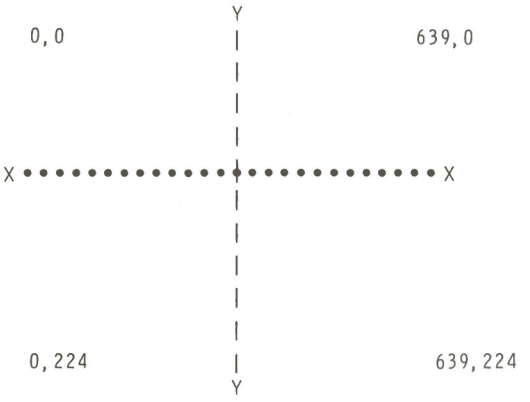



**FIGURE 7.1**
**X,Y Coordinates of the Four-Corner Points**

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<br>**1** — Sets H-19 Graphics mode |
| Reverse Video | **0** — Clears H-19 reverse video<br>**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           'graphics off and 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 from the screen at the specified row (line) and column.

Format: `x = SCREEN(row,col [,z])`

**x** is a numeric variable receiving the ASCII value returned.

**row** is a valid numeric expression returning an unsigned integer in the range one to 25.

**col** is a valid numeric expression returning an unsigned integer in the range one to 80.

**z** is a valid numeric expression returning a Boolean result.

Action:

The ASCII value of the character at the specified coordinates is stored in the numeric variable. If the optional parameter <z> is given and is not zero, the number returned, when reduced by modulo 8, is the color attribute of the character.

Suppose for example the letter O was in row one, column one.

```
10 X=screen (1,1)
20 PRINT X
RUN
79
Ok
```

The number 79 is the ASCII equivalent of the letter O. To verify this, you could either use the ASCII table in Appendix C, or you could use the CHR$ function in the following manner:

```
PRINT CHR$(79)
O
Ok
```

# PLOTTING COORDINATES

## The Video Screen

Rule:

1. Any values entered outside the range for row (1-25) or the range for column (1-80) will result in an `Illegal Function Call` error.

Examples:

```
100 X = SCREEN (10,10)    'If the character at 10,10 is
                          'A then return 65.

110 X = SCREEN (1,1,1)    'Return the color attribute of
                          'the character in the upper left
                          'hand corner of the screen.
```

```
10 COLOR 4,7       'Make foreground red and background white
20 CLS: PRINT "H" 'Print H in top left corner
30 X=SCREEN (1,1,1) 'Use SCREEN function to determine attribute of H
40 PRINT X
RUN
 100
Ok
```

This example prints an H in the top left corner of the screen. Then the screen function is used to determine the attribute of H. To interpret the result you must use the MOD operator (see Page 5.22) with the number 8, to reduce the result to one of the valid color numbers as shown below. Four is the color red.

```
PRINT 100 MOD 8
 4
Ok
```

# 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
```

## 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.

**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 ,,` | Position and cursor visibility remain unchanged. |
| `40 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.

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.

# 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.

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.

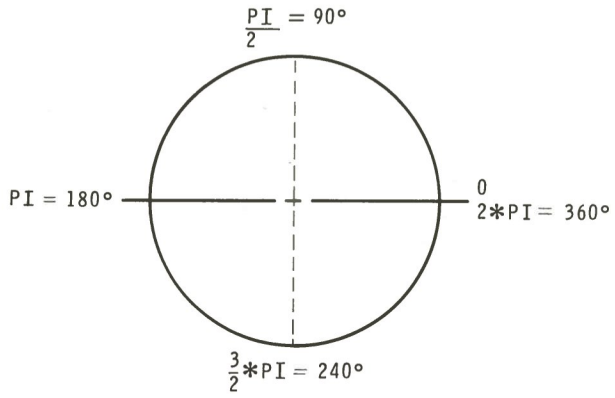



**Figure 8.1.**
**Angles of a Circle**

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).

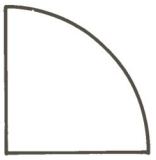
# ADVANCED COLOR GRAPHICS

## LINE, CIRCLE and PAINT Statements

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 .4375 and will give a visual circle assuming a standard monitor screen aspect ratio of 7/16.

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

```
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.

# ADVANCED COLOR GRAPHICS

## LINE, CIRCLE and PAINT Statements

### 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 [,border attribute]]`

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.

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<"stringexpression">`

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),arrayname`
Format: `PUT(X1,Y1),array[,actionverb]`

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<"stringexpression">`

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

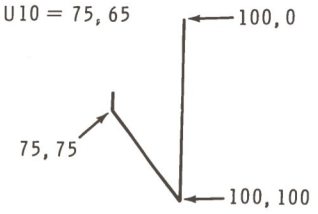
Absolute and Relative Moves

**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$="M75,75 U10"
40 DRAW B$
```



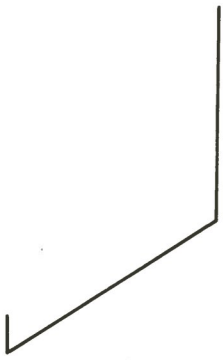

This example tells BASIC to draw a line from (100,0) to (100,100) and from that point (100,100 ) draw a line to the absolute address of (75,75) then draw a line up 10 points (75,65). If you inserted a "+" sign in line 30:

**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.

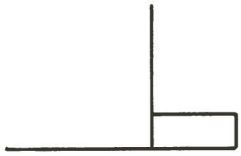


**A[<n>]** Set angle n. n may range from zero to three, where zero is zero degrees, one is 90, two is 180, and three is 270. Figures rotated 90 or 270 degrees are scaled so that they will appear the same size as with zero or 180 degrees on a monitor screen with the standard aspect ratio of 4/3. In the 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

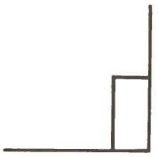
If an angle command A0 was added to line 30,

**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"**

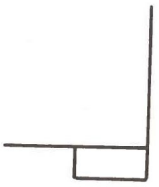
the box would be rotated 90 degrees and appear as shown below.



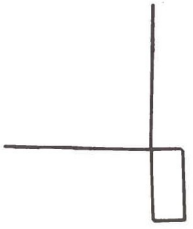
If A2 was substituted in line 30,

**30 B$="A2 U10 R50 D10 L50"**

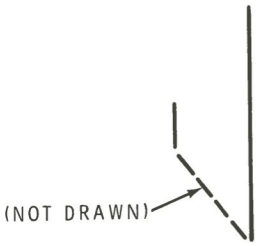
the box would be rotated 180 degrees and appear as shown below.



Finally, A3 would cause the box to be rotated 270 degrees and appear as shown below.



## PREFIX COMMANDS



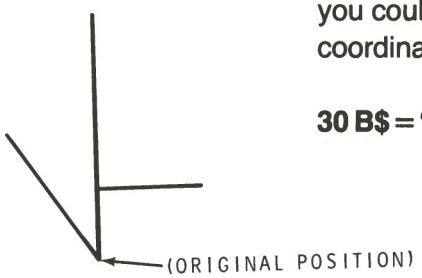

B Move but don't plot any points. The B command permits you to move to a different location without plotting any points. If you inserted a B in front of a movement command in line 30:

**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.

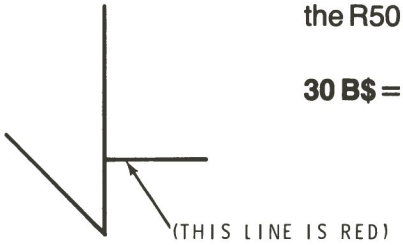
**30 B$ = "NM75,75U10R50"**




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

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 (#).

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 hold 193 bytes. The next question is, how many bytes per element will be in this array? If you declared the array to be interger, (%) 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
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 string
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
```

## Z-BASIC Summary Program

```
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 D0R=40' Size of door
630 R00F$="E50R120F50"
640 LSIDE$="BL100U65XR00F$;" ' Left side of house
650 DRAW"BM+100,23D65U20" ' Right side of house
660 DRAW"BM300,200U=D0R;R=D0R;D=D0R;L=D0R;XLSIDE$;"
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. |

# 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$ | 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$ | 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. |

| | |
|---|---|
| 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. |
| 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 ordinal of specified character. |
| SPC | Prints blanks on the terminal or the line printer. |
| TAB | Spaces to a position of the terminal or line printer. |
| VARPTR | Returns an address value which can be used to locate where the variable <variable name> is stored in memory. |
| WIDTH L PRINT | 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.

## 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. |