BASIC–PLUS
Language Manual

Order No. AA–2623D–TC

digital software

BASIC–PLUS
Language Manual

Order No. AA–2623D–TC

December 1981

This manual describes the BASIC–PLUS language and the use of this language on the RSTS/E operating system.

OPERATING SYSTEM AND VERSION:	RSTS/E	V7.1
SOFTWARE VERSION:	BASIC–PLUS	V7.1

digital equipment corporation, maynard, massachusetts

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license, and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

Copyright © 1975, 1976, 1979, and 1981 Digital Equipment Corporation

The postage-paid READER'S COMMENTS form on the last page of this document requests your critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	VT	IAS
DECUS	DECsystem–10	MASSBUS
DECnet	DECSYSTEM 20	PDT
PDP	DECwriter	RSTS
UNIBUS	DIBOL	RSX
VAX	EduSystem	VMS

Commercial Engineering Publications typeset this manual using DIGITAL's TMS–11 Text Management System.

Contents

Page

Preface

Part I Developing BASIC–PLUS Programs

Chapter 1 Introduction to BASIC–PLUS

1.1 Programming in BASIC–PLUS 1–2
1.2 Commands and Statements 1–3
1.2.1 BASIC–PLUS Terminal Session 1–3
1.2.1.1 Creating the Program 1–3
1.2.1.2 Listing the Program 1–4
1.2.1.3 Running the Program 1–4
1.2.1.4 Saving the Program 1–5
1.2.1.5 Retrieving and Changing the Program 1–5
1.2.1.6 Saving the Modified Program 1–6
1.2.2 A Sample Program 1–7

Chapter 2 BASIC–PLUS and RSTS/E

2.1 Terms 2–1
2.2 The BASIC–PLUS Run-time System 2–2
2.3 Types of RSTS/E Systems 2–3
2.4 Using BASIC–PLUS on a BASIC–PLUS System 2–3
2.5 Using BASIC–PLUS on Other RSTS/E Systems 2–3
2.5.1 DCL Systems 2–4
2.5.2 Other Non–BASIC–PLUS Systems 2–4

Chapter 3 Overview of BASIC–PLUS Program Development

3.1 Source and Translated Programs 3–1
3.2 The Current Program 3–2
3.3 Command and Statement Summary 3–2

Chapter 4 Creating and Running a BASIC–PLUS Program

4.1 Creating a Program 4–1
4.1.1 The NEW Command 4–1
4.1.2 Entering the Program 4–2
4.1.2.1 Correcting Typing Errors 4–2
4.1.2.2 Correcting Syntax Errors 4–3
4.1.2.3 Program Translation 4–3
4.1.2.4 The TEMPnn.TMP File 4–3
4.2 Calling an Existing Program – The OLD Command 4–4
4.3 Displaying a Program – The LIST Command 4–5
4.4 Running a Program – The RUN Command 4–6

4.5 Saving a Source Program – The SAVE Command 4–7
4.6 Saving a Compiled Program – The COMPILE Command 4–8
4.7 Getting Information 4–9
4.7.1 Listing Files in a Directory – The CATALOG Command 4–9
4.7.2 Displaying a Program's Length in Memory – The LENGTH Command 4–10
4.7.3 Displaying the Scale Factor – The SCALE Command 4–11
4.8 Changing Modes – EXTEND and NOEXTEND 4–12

Chapter 5 Modifying BASIC–PLUS Programs

5.1 Editing a BASIC–PLUS Program 5–1
5.1.1 Entering New Program Statements 5–1
5.1.2 Using the DELETE Key and CTRL/U 5–2
5.1.3 The DELETE Command 5–2
5.2 Changing a Program's Name or File Specification 5–3
5.2.1 The RENAME Command 5–3
5.2.2 The NAME–AS Statement 5–4
5.3 Replacing a Saved Program – The REPLACE Command 5–6
5.4 Deleting a Saved Program 5–6
5.4.1 The UNSAVE Command 5–6
5.4.2 The KILL Statement 5–7
5.5 Merging Programs – The APPEND Command 5–8

Chapter 6 Immediate Mode and Program Debugging

6.1 Immediate Mode 6–2
6.1.1 Immediate Mode Examples 6–2
6.1.2 Variable Assignments 6–3
6.1.3 Limitations of Immediate Mode 6–3
6.2 Background for Debugging – The Ready State 6–4
6.2.1 Entering the Ready State 6–5
6.2.2 Program Status 6–6
6.2.3 Possible Actions 6–7
6.2.4 Summary 6–8
6.3 Debugging in Immediate Mode 6–8
6.3.1 The STOP Statement 6–8
6.3.2 The CONT Command 6–9
6.3.3 The CCONT Command 6–9
6.3.4 The GOTO Statement 6–9
6.3.5 Debugging Example Using STOP and CONT 6–10
6.4 Halting and Checking Execution – CTRL/C and PRINT LINE 6–12
6.5 Controlling Terminal Output 6–13
6.5.1 Stopping Output with CTRL/O 6–13
6.5.2 Suspending and Resuming Output with CTRL/S and CTRL/Q 6–13
6.5.3 Suspending and Resuming Output with the NO SCROLL Key 6–14

6.6 Getting a Cross-Reference Listing – The BPCREF Program 6–14
6.6.1 Running BPCREF 6–14
6.6.2 Output Listing Contents 6–16
6.6.2.1 Header Line 6–17
6.6.2.2 Cross-Reference Table 6–17
6.6.2.3 Statistical Data 6–18
6.6.2.4 Optional List of Suspect Line Numbers and Variables 6–18
6.6.2.5 Global and Local Listings 6–19
6.6.3 Error Messages 6–20

PART II Elementary Language Features

Chapter 7 Building a Program

7.1 Sample BASIC–PLUS Program 7–1
7.2 Parts of a Program 7–1
7.2.1 Line Numbers 7–1
7.2.2 Statements 7–3
7.3 Line and Statement Formats 7–4
7.3.1 Multi-Statement Lines 7–4
7.3.2 Multi-Line Statements 7–5
7.3.3 Spaces and Tabs 7–6
7.4 Remarks and Comments 7–6
7.5 EXTEND and NOEXTEND Modes 7–9
7.5.1 Changing Modes 7–9
7.5.2 EXTEND and NOEXTEND Program Formats 7–10
7.5.2.1 Variable and Function Names 7–11
7.5.2.2 Spaces and Tabs 7–11
7.5.2.3 Continuation Lines 7–12
7.5.2.4 Comments 7–12

Chapter 8 Building Statements

8.1 BASIC–PLUS Character Set 8–1
8.2 Keywords 8–1
8.3 Data 8–2
8.3.1 Constants 8–3
8.3.1.1 Real Constants 8–3
8.3.1.2 Integer Constants 8–4
8.3.1.3 String Constants 8–4
8.3.2 Variables 8–5
8.3.2.1 Naming Variables 8–6
8.3.2.2 Real Variables 8–6
8.3.2.3 Integer Variables 8–6
8.3.2.4 String Variables 8–7
8.4 Expressions 8–7
8.4.1 Arithmetic Expressions 8–8

8.4.2 String Expressions 8–10
8.4.3 Relational Expressions 8–10
8.4.3.1 Numeric Relational Expressions 8–11
8.4.3.2 String Relational Expressions 8–12
8.4.4 Logical Expressions 8–14
8.4.5 How Expressions Are Evaluated 8–16

Chapter 9 Elementary Statements and Features

9.1 LET Statement 9–2
9.2 Introduction to Programmed Input and Output 9–3
9.2.1 PRINT Statement 9–3
9.2.1.1 Printing Numbers and Character Strings 9–4
9.2.1.2 Formatting the Output 9–5
9.2.2 INPUT Statement 9–7
9.2.3 READ and DATA Statements 9–9
9.2.4 RESTORE Statement 9–12
9.3 Unconditional Branch, GOTO Statement 9–13
9.4 Conditional Branch, IF–THEN and IF–GOTO Statements 9–13
9.5 Program Loops 9–15
9.5.1 FOR and NEXT Statements 9–17
9.5.2 WHILE and NEXT Statements 9–22
9.5.3 UNTIL and NEXT Statements 9–23
9.6 Subscripted Variables and the DIM Statement 9–24
9.7 Mathematical Functions 9–27
9.7.1 Sign Function, SGN(X) 9–27
9.7.2 Integer Function, INT(X) 9–28
9.7.3 Random Number Function, RND 9–29
9.7.4 RANDOMIZE Statement 9–30
9.8 User-Defined Functions 9–31
9.9 Subroutines 9–35
9.9.1 GOSUB Statement 9–36
9.9.2 RETURN Statement 9–36
9.9.3 Nesting Subroutines 9–36
9.10 END Statement 9–37
9.11 STOP Statement 9–38

PART III Advanced Language Features

Chapter 10 Strings and String Functions

10.1 Strings 10–1
10.1.1 String Constants 10–1
10.1.2 String Variables 10–1
10.1.3 Subscripted String Variables 10–2
10.1.4 String Size 10–3
10.1.5 Relational Operators 10–3

10.2 ASCII String Conversions, CHANGE Statement 10–3
10.3 String Input. 10–6
10.3.1 READ and DATA Statements 10–6
10.3.2 INPUT Statement 10–7
10.3.3 INPUT LINE Statement. 10–7
10.4 String Output 10–9
10.5 String Functions 10–9
10.5.1 CVT$$ Function. 10–12
10.5.2 XLATE Function 10–17
10.6 User-Defined String Functions 10–18
10.7 String Arithmetic 10–19
10.7.1 String Arithmetic Precision 10–21
10.7.2 Combining String Arithmetic Functions 10–24

Chapter 11 Integer and Floating-Point Operations

11.1 Introduction to Integers 11–1
11.2 Internal Integer Format. 11–2
11.3 Integer Constants and Variables. 11–4
11.4 Integer Arithmetic 11–4
11.5 Integer I/O 11–5
11.6 User-Defined Integer Functions 11–5
11.7 Logical Operations on Integer Data 11–6
11.7.1 The Logical Values –1% and 0% 11–6
11.7.1.1 Relational Expressions. 11–6
11.7.1.2 Logical Expressions 11–7
11.7.2 Other Logical Values 11–8
11.7.3 How BASIC–PLUS Performs Logical Operations. 11–8
11.7.4 Programming Applications 11–11
11.7.4.1 Bit Tests 11–11
11.7.4.2 Setting or Clearing Bits 11–11
11.7.4.3 Bit Masks 11–12
11.8 Floating-Point Arithmetic 11–13
11.9 Mixed-Mode Arithmetic 11–14
11.10 Scaled Arithmetic 11–15
11.10.1 The Scale Factor 11–15
11.10.2 The SCALE Command 11–17

Chapter 12 Matrix Manipulation

12.1 Array Storage. 12–1
12.2 MAT READ Statement 12–2
12.3 MAT PRINT Statement 12–2
12.4 MAT INPUT Statement 12–4
12.5 Matrix Initialization Statements. 12–6
12.6 Matrix Calculations 12–7
12.6.1 Matrix Operations. 12–7
12.6.2 Matrix Functions 12–8

Chapter 13 Advanced Statements and Features

13.1 DEF* Statement, Multiple-Line Function Definitions 13–1
13.2 ON–GOTO Statement 13–4
13.3 ON–GOSUB Statement 13–4
13.4 IF–THEN–ELSE Statement 13–5
13.5 Conditional Termination of FOR Loops 13–7
13.6 Statement Modifiers 13–10
13.6.1 IF Statement Modifier 13–10
13.6.2 UNLESS Statement Modifier 13–11
13.6.3 FOR Statement Modifier 13–11
13.6.4 WHILE Statement Modifier 13–12
13.6.5 UNTIL Statement Modifier 13–13
13.6.6 Multiple Statement Modifiers 13–14
13.7 Error Handling 13–14
13.7.1 ON ERROR GOTO Statement 13–15
13.7.2 RESUME Statement 13–15
13.7.3 Disabling the Error Handling Subroutine 13–16
13.7.4 The ERL Variable 13–17
13.8 System Functions 13–18
13.9 SLEEP and WAIT Statements 13–20
13.10 CHAIN Statement 13–22

PART IV Data Handling

Chapter 14 Overview of Data Handling

14.1 Files and Devices 14–2
14.2 Accessing a File or Device from a Program 14–2
14.2.1 Opening the File 14–2
14.2.2 Reading and Writing Data 14–3
14.2.3 Closing the File 14–3
14.3 BASIC–PLUS File Organizations 14–4
14.3.1 Formatted ASCII Files 14–4
14.3.2 Virtual Array Files 14–4
14.3.3 Block I/O Files 14–5
14.4 Choosing an I/O Method 14–5
14.5 OPEN Statement 14–6
14.5.1 Forms of the OPEN Statement 14–7
14.5.2 File-Structured and Non-File-Structured Devices 14–10
14.5.3 OPEN Statement Options 14–10
14.5.3.1 RECORDSIZE Option 14–10
14.5.3.2 CLUSTERSIZE Option 14–12
14.5.3.3 FILESIZE Option 14–14
14.5.3.4 MODE Option 14–14
14.6 CLOSE Statement 14–15
14.6.1 CLOSE 14–15
14.6.2 CLOSE with a Negative Channel Number 14–15

14.7 NAME–AS Statement (File Protection and Renaming) 14–16
14.8 KILL Statement 14–17

Chapter 15 Formatted ASCII Input and Output

15.1 PRINT Statement 15–1
15.1.1 Printing Data to a File or Device (Formatted ASCII Output) . 15–3
15.1.2 PRINT–USING Statement 15–4
15.1.2.1 Exclamation Point 15–4
15.1.2.2 String Field 15–5
15.1.2.3 Numeric Field 15–5
15.1.2.4 Asterisks 15–6
15.1.2.5 Exponential Format 15–6
15.1.2.6 Trailing Minus Sign 15–6
15.1.2.7 Dollar Signs 15–7
15.1.2.8 Commas 15–7
15.1.2.9 Insufficient Format 15–7
15.1.2.10 Format Too Large 15–8
15.1.2.11 Formatting and Literal Characters 15–8
15.1.3 MAT PRINT Statement 15–9
15.1.4 PRINT Functions 15–10
15.2 INPUT Statement 15–11
15.2.1 Reading Data from a File or Device 15–13
15.2.2 Opening Your Terminal as an I/O Channel 15–15
15.2.3 INPUT LINE Statement 15–15
15.2.4 MAT INPUT Statement 15–16
15.3 Formatted ASCII Examples 15–17

Chapter 16 Virtual Arrays

16.1 Virtual Array DIM Statement 16–2
16.2 Virtual Array String Storage 16–2
16.3 Opening and Closing a Virtual Array File 16–3
16.3.1 Preextending a Virtual Array 16–4
16.3.2 Closing a Virtual Array File 16–5
16.4 Virtual Array Programming Conventions 16–5
16.4.1 Virtual Array Storage 16–5
16.4.2 Translation of Array Subscripts into File Addresses 16–6
16.4.3 Access to Data in Virtual Arrays 16–10
16.4.4 Allocating Disk Storage to Virtual Array Files 16–11
16.4.5 Simultaneous Access of a Virtual Array 16–12
16.5 Programming Examples 16–12

Chapter 17 Block I/O

17.1 Opening a Block I/O File 17–3
17.1.1 STATUS Variable 17–3
17.1.2 BUFSIZ Function 17–3

17.2 Closing a Block I/O File 17–4
17.3 Reading and Writing Data – The GET and PUT Statements 17–5
17.3.1 BLOCK Option 17–6
17.3.2 RECORD Option 17–7
17.3.3 COUNT Option 17–7
17.3.4 USING Option 17–8
17.3.5 RECOUNT Variable. 17–9
17.3.6 Extending Disk Files 17–9
17.3.7 Alternate Buffer I/O 17–10
17.4 Accessing the I/O Buffer 17–11
17.4.1 FIELD Statement 17–12
17.4.2 The LSET and RSET Statements 17–14
17.4.3 Differences between the LET Statement and the LSET/RSET Statements. 17–15
17.5 Converting Numeric Data – CVT Functions and SWAP% 17–16
17.5.1 CVT Conversion Functions 17–16
17.5.2 SWAP% Function 17–17
17.6 Block I/O Examples. 17–18
17.7 UNLOCK Statement 17–21

Appendix A Language Summary

A.1 Summary of Variable Types A–1
A.2 Summary of Operators A–2
A.3 Summary of Functions and Variables A–2
A.4 Summary of Statements A–7
A.4.1 Statements A–8
A.4.2 Statement Modifiers. A–10
A.5 Reserved Keywords A–20

Appendix B Command Summary

Appendix C Error Messages

C.1 Interpretation of Error Messages C–1
C.2 The "?Program Lost–Sorry" Error C–14
C.2.1 Checksum Error on a .BAC File C–15
C.2.2 Unrecoverable Disk Error Reading a .BAC File C–15
C.2.3 Incorrect .BAC File Size C–16
C.2.4 Unmatched Version Numbers C–16
C.3 Reporting Software Problems C–16

Appendix D Character Set

Appendix E Hints for BASIC–PLUS/BASIC–PLUS–2 Compatibility

Appendix F Programming Hints

F.1 Optimizing BASIC–PLUS Programs F–1
F.1.1 Optimizing Statement Formats F–1
F.1.2 Using Variables Efficiently F–2
F.1.3 Using Constants Efficiently F–3
F.1.4 Statement Modifiers F–4
F.1.5 Optimizing Statement Structure F–4
F.2 Decreasing Disk Access Time F–5
F.3 Manipulating Strings Efficiently F–6
F.4 Converting Numeric Data F–6
F.5 Accessing Algorithm for Virtual Arrays F–9

Glossary

Index

Figures

6–1 Sample Cross-Reference Listing 6–17
7–1 Sample BASIC–PLUS Program 7–2
7–2 NOEXTEND Format 7–10
7–3 EXTEND Format 7–10
9–1 Nesting Techniques 9–20
9–2 Array Structure 9–25
11–1 Internal Integer Format 11–2
F–1 CVT Conversion of Integer Data F–7
F–2 CVT Conversion of Two-Word Floating-Point Data F–8
F–3 CVT Conversion of Four-Word Floating-Point Data F–8
F–4 Virtual Array Accessing Algorithm F–10

Tables

3–1 BASIC–PLUS Commands 3–2
3–2 BASIC–PLUS Statements 3–3
3–3 Control Characters and Terminal Keys 3–4
6–1 Program Status in the Ready State 6–8
6–2 BPCREF Command Formats 6–15
6–3 BPCREF Command Switches 6–16
6–4 BPCREF Error Messages 6–21
8–1 BASIC–PLUS Data Types 8–3
8–2 Arithmetic Operators 8–9
8–3 Numeric Relational Operators 8–11
8–4 String Relational Operators 8–12
8–5 Logical Operators 8–15
8–6 Truth Values for Logical Operations 8–15
8–7 Operator Precedence 8–16
9–1 Mathematical Functions 9–28
10–1 String Functions 10–9
10–2 Optional String Arithmetic Functions 10–20
10–3 Precision Values in PROD$, QUO$, and PLACE$ Functions 10–24
11–1 Truth Values for Logical Operations 11–9
13–1 System Functions 13–18
13–2 TIME$ String Examples 13–20
14–1 OPEN Statement Errors 14–9
14–2 Default Device Buffer Sizes 14–11
14–3 Use of RECORDSIZE 14–11
16–1 Virtual Array Storage 16–6
17–1 STATUS Variable 17–4
17–2 Device Record Characteristics 17–6
17–3 CVT Conversion Functions 17–16
A–1 Reserved Keywords A–20
B–1 BASIC–PLUS Commands B–1
B–2 Control Characters and Terminal Keys B–5
C–1 Severity Standard in Error Messages C–2
C–2 Special Abbreviations for Error Descriptions C–2
C–3 Nontrappable Errors in Recoverable Class C–3
C–4 User-Recoverable Error Messages C–4
C–5 Nonrecoverable Error Messages C–10
D–1 Special Symbols and Keys D–1
D–2 ASCII Character Codes D–3

Preface

This manual describes the BASIC–PLUS programming language as implemented for the RSTS/E operating system.

BASIC–PLUS is an easy language for beginning programmers to learn. It also provides many advanced features for experienced programmers. This manual is intended for all BASIC–PLUS programmers but is organized to help the beginner.

Manual Structure

This manual has four parts, each containing several chapters. The manual also has six appendixes.

Part I (Chapters 1 through 6) introduces BASIC–PLUS, explains the relationship between BASIC–PLUS and the RSTS/E operating system, and describes the commands and features used to write, run, modify, and debug programs.

Part II (Chapters 7 through 9) describes elementary BASIC–PLUS features. It describes EXTEND and NOEXTEND program formats and BASIC–PLUS data types and expressions. It also introduces BASIC–PLUS statements such as LET, GOTO, INPUT and PRINT, and features such as mathematical functions and subroutines. You can solve many programming problems using the statements and features described in Part II.

Part III (Chapters 10 through 13) describes the advanced features of BASIC–PLUS, including string functions, statement modifiers, error handling, and matrix manipulation statements.

Part IV (Chapters 14 through 17) describes BASIC–PLUS data handling: formatted ASCII, virtual array, and block I/O files.

The appendixes:

- Review the BASIC–PLUS language in summary form
- List error messages that BASIC–PLUS users can encounter
- List hints for writing BASIC–PLUS programs that are compatible with BASIC–PLUS–2
- Provide programming hints for the advanced user

Related Documents

The *RSTS/E Primer* and the *Introduction to BASIC* introduce the RSTS/E operating system and the BASIC–PLUS programming language.

The *RSTS/E System User's Guide* describes how to use RSTS/E system programs and how to work with files and devices.

The *RSTS/E Programming Manual* describes advanced BASIC–PLUS programming techniques, including the use of SYS system function calls and device-dependent features.

See the *RSTS/E Documentation Directory* for more information on RSTS/E manuals.

Conventions

This manual uses the following conventions:

Color	Red print shows what you type in examples.
UPPERCASE	In statement descriptions, items in capital letters (LET, IF, and THEN, for example) must be typed exactly as shown. They are BASIC–PLUS keywords.
< >	Angle brackets enclose essential elements of the statement or command being described. For example, you must specify a variable and an expression in the LET statement: [LET] <variable> = <expression>
{ }	Braces indicate a required choice of one element among two or more possibilities. For example: IF <condition> { THEN <statement> / THEN <line number> / GOTO <line number> }
[ ]	Square brackets indicate an optional statement element or a choice of one element among two or more optional elements. For example: IF <condition> { THEN <statement> / THEN <line number> / GOTO <line number> } [ELSE <statement> / ELSE <line number>]
CTRL/x	This symbol indicates a control key combination, such as CTRL/U or CTRL/O. To enter a control key combination, hold the CTRL key down while you press the indicated key.
^	The circumflex is a control character's echo. For example, when you enter CTRL/U, the system displays "^U" on the terminal. The circumflex also means exponentiation, the mathematical operation that raises a number to a power.

▒ This symbol represents the cursor, the blinking white line or rectangle that marks the current position on your terminal screen.

(LF) This symbol represents the LINE FEED key on your terminal.

(RET) This symbol represents the RETURN key on your terminal.

Most examples in this manual do not show the RETURN key symbol. It appears in the introductory examples to help you get started but does not appear in the rest of the manual. If you try examples at your terminal, always press the RETURN key when you finish typing a command, immediate mode statement, or line in a program unless the example indicates otherwise.

The programming examples in this manual shows programs as they appear when you finish entering them at the terminal, or when you display them on your screen with the LISTNH command. (LISTNH displays the program currently in memory.) In addition, most examples contain EXTEND mode features, such as the ampersand/RETURN key combination for line continuation. If you get error messages when you try the examples, your system probably uses NOEXTEND as the default mode. To correct the problem, enter the EXTEND command or enter an EXTEND statement at the beginning of the program. Both put BASIC–PLUS in EXTEND mode.

This manual uses the following terms:

BASIC–PLUS	Means the BASIC–PLUS language, the BASIC–PLUS run-time system (the system software that accepts and executes BASIC–PLUS programs), or both, depending on the usage.
System	Means the RSTS/E operating system.
Print and Type	BASIC–PLUS prints on the terminal; you type at the keyboard.
Statement	A single BASIC–PLUS language instruction identified by one or more BASIC–PLUS language keywords.
Program	A series of instructions written in BASIC–PLUS or another programming language.
Command	An instruction that causes BASIC–PLUS or a system program to perform some operation immediately. Commands are not part of a program and are not preceded by a line number.

PART I
Developing BASIC–PLUS Programs

Chapter 1
Introduction to BASIC–PLUS

BASIC–PLUS is a version of BASIC available on RSTS/E. (BASIC, which stands for Beginner's All-purpose Symbolic Instruction Code, is a registered trademark of Dartmouth College.)

BASIC–PLUS is both a programming environment and a programming language. The programming environment is a set of commands for working with programs; the programs themselves are composed of statements written in the BASIC–PLUS language.

The BASIC–PLUS programming environment is one of several command environments available on RSTS/E. It is the main command environment on many, but not all RSTS/E systems. You can recognize BASIC–PLUS by its "Ready" prompt.

The BASIC–PLUS programming language has both standard features and optional features. With standard features, you can solve a wide variety of programming problems. Optional features, on the other hand, provide more specialized capabilities. They let you:

- Do more precise calculations than you can with standard features
- Work with matrices using special statements that operate on an entire matrix
- Print data in special formats
- Take advantage of built-in mathematical functions

This manual describes all BASIC–PLUS features and tells you which are optional.

1.1 Programming in BASIC–PLUS

BASIC–PLUS is an interactive programming environment. After you enter each line in your program, BASIC–PLUS checks to make sure its format and syntax are correct. If they are correct, BASIC–PLUS translates the line into code that the system can execute. If the format and syntax are not correct, BASIC–PLUS prints an error message on your terminal and lets you reenter the line correctly. Because BASIC–PLUS translates each line as you enter it, you can run a program right after you finish entering it. If the program does not work correctly, you can easily modify and rerun it. The terminal session in this chapter shows interactive use of BASIC–PLUS.

Immediate mode, another BASIC–PLUS feature, lets you execute statements without writing a complete program. When you enter statements into a program, you use line numbers. To enter an immediate mode statement, omit the line number. BASIC–PLUS executes the statement after you enter it instead of storing it in a program. Immediate mode operation is especially useful in two areas: debugging programs and performing simple calculations. For example, in immediate mode you can examine or change variables in the current program or use BASIC–PLUS as a "desktop calculator."

The following example shows how immediate mode works. To try the example, log in and then type the portions of the example in red. If you see the "Ready" system prompt, you are in the BASIC–PLUS environment and can proceed. If your system prompt is not "Ready," you must enter the BASIC–PLUS environment.

If you are on a DCL system, type the following command to enter BASIC–PLUS. You can recognize DCL by its "$" prompt.

```
$ BASIC/BPLUS<RET>
```

If your system prompt is a different character, such as ">" or ".", type the following command to enter BASIC–PLUS:

```
>RUN $SWITCH<RET>
Keyboard Monitor to switch to? BASIC<RET>
```

The system displays the Ready prompt. Now type:

```
PRINT 54+46<RET>
```

BASIC–PLUS adds these two numbers and displays the result:

```
 100
```

If you want to try the terminal session in the next section, stay in BASIC–PLUS. Otherwise, type RUN $SWITCH and press the RETURN

key twice to return to your system's main command environment or use the BYE command to log out. (BYE is described in the *RSTS/E System User's Guide*.)

1.2 Commands and Statements

BASIC–PLUS has both commands and statements. In general, you use BASIC–PLUS commands to develop programs – that is, to create, modify, and run them. BASIC–PLUS statements, on the other hand, make up the BASIC–PLUS language. You use statements in programs to input data, do computations, and print results. Part I of this manual explains how to work with programs; Parts II, III, and IV describe the BASIC–PLUS programming language.

1.2.1 BASIC–PLUS Terminal Session

The following example introduces the BASIC–PLUS commands. It shows you how to create, display, save, run, retrieve, change, and replace a simple BASIC–PLUS program. Do not be concerned about what the statements in the program mean; they will be explained later. Just follow the steps below. Type the portions of the example shown in red.

The sample program computes the factorial of a number. (The factorial of a number n is the product of all positive integers from 1 to n.) When run, the program asks you for a number from 1 to 15 and prints its factorial.

This terminal session assumes that you are logged in. Although the example shows all commands and statements in uppercase, you can enter them in either upper- or lowercase. Once again, if you see the Ready prompt on your terminal, you can proceed. Otherwise, use the SWITCH program or the BASIC/BPLUS command to enter BASIC–PLUS.

1.2.1.1 Creating the Program

To create a BASIC–PLUS program, you first use the NEW command to name the program, and then you enter the program statements. The sample program name is FACTOR. Press the RETURN key after each program statement. The RETURN key ends the statement and moves the cursor to a new line.

```
NEW<RET>
New file name--FACTOR<RET>

Ready

10 PRINT "Enter a number between 1 and 15"<RET>
20 PRINT "and this program will calculate its factorial."<RET>
30 INPUT N<RET>
40 LET F=1<RET>
50 FOR X=1 TO N<RET>
60 LET F=F*X<RET>
70 NEXT X<RET>
80 PRINT "The factorial of";N;"is";F<RET>
90 END<RET>
```

Use the DELETE key to correct typing mistakes as you enter the statements. You can use the DELETE key only before you press the RETURN key; if you do not notice your mistake until after you press the RETURN key, retype the line number and the corrected statement. BASIC–PLUS replaces the incorrect line in your program with the new line.

Suppose, for example, that you forget the first quotation mark in line 10, which is required for the statement to be correct. The system prints an error message followed by the Ready prompt. Reenter line 10 correctly before typing other program statements.

```
10 PRINT Enter a number between 1 and 15"RET
? Syntax error at line 10

Ready

10 PRINT "Enter a number between 1 and 15"RET
```

1.2.1.2 Listing the Program

Although BASIC–PLUS checks for errors in the syntax or format of your BASIC–PLUS statements as you type them, it does not detect other types of errors. For example, BASIC–PLUS does not print an error message if you misspell a word inside quotation marks or if you enter a statement that is formatted correctly but does not produce the results you want when you run the program.

Looking at your program is one way to check for correctness. Display the program on your terminal by typing:

```
LISTRET
```

If any statements contain typing mistakes, correct them. Retype the line number and statement, and press the RETURN key. After making corrections, list the program again.

1.2.1.3 Running the Program

You are now ready to run the program. Type:

```
RUNRET
```

The program prints a header line and prompts you for input with a question mark. Enter the value shown.

```
FACTOR  01:50 PM        03-Feb-81
Enter a number between 1 and 15
and this program will calculate its factorial.
? 9RET
The factorial of 9 is 362880

Ready
```

If you wish, run the program again and enter a different number. If you enter a number larger than 9, the result will be greater than six digits. BASIC–PLUS will print the result in exponential format, a mathematical "shorthand" used for large numbers. For example:

```
RUN↵
FACTOR  01:51 PM        03-Feb-81
Enter a number between 1 and 15
and this program will calculate its factorial.
? 12↵
The factorial of 12 is  .479002E 9

Ready
```

1.2.1.4 Saving the Program

The sample program is currently in memory. The program will be deleted from memory if you create a new program with NEW, retrieve an existing BASIC–PLUS program with the OLD command, run another program, or log out. Use the SAVE command to save the sample program in a disk file in your user account. The program is saved with the name FACTOR. Type:

```
SAVE↵
```

SAVE copies the program; it does not remove the program from memory. Use the SAVE command to save a copy of a program after you create it.

1.2.1.5 Retrieving and Changing the Program

Now you are going to change the sample program. Before making any changes, however, use the OLD command to retrieve a "clean" copy of the program from disk and place it in memory. While not required, this step is good programming practice because it clears your memory area. Clearing memory ensures that your program takes up the smallest possible amount of memory space. Type:

```
OLD FACTOR↵
```

As mentioned previously, if you use the sample program to calculate the factorial of a number greater than 9, it prints the result in exponential format. You are going to change the PRINT statement in line 80 and add a new line. These changes cause the program to print large numbers in a more readable format. Type the following lines:

```
80 PRINT "The factorial of";N;"is ";↵
85 PRINT NUM1$(F)↵
```

After you type the new lines, use the LIST command to display the program on your terminal and then run the program one or more times. For example:

```
RUN(RET)
FACTOR  02:03 PM        03-Feb-81
Enter a number between 1 and 15
and this program will calculate its factorial.
? 12(RET)
The factorial of 12 is 479002000

Ready
```

NOTE

BASIC–PLUS has two types of math packages: single-precision and double-precision. Depending on which math package is installed on a system, BASIC–PLUS calculates numbers to six or fifteen significant digits. The two math packages are described later in this manual.

This example was run on a BASIC–PLUS system with the single-precision math package. If your system has the double-precision math package, it displays the following value for the factorial of 12:

```
479001600
```

1.2.1.6 Saving the Modified Program

The version of FACTOR that is currently in memory contains the changes you made, but the version of FACTOR that you saved on disk with the SAVE command still contains the old program.

To replace the old version of FACTOR with the current version, type:

```
REPLACE(RET)
```

Use the REPLACE command often during a BASIC–PLUS programming session to save changes to a program as you make them. Certain BASIC–PLUS commands and all system commands entered from the BASIC–PLUS environment delete your current program from memory. Thus, you can lose the latest version of your program if you do not use REPLACE. (Later chapters explain which commands delete your current program.)

If you want to keep the old version of FACTOR, you can save the new version with a new name. For example:

```
SAVE FACT2
```

This SAVE command creates a new disk file in your account that contains the modified FACT2 program. From now on, always use the name FACT2 to work with this version of the program.

You are finished with this terminal session. Use the BYE command to log out or the SWITCH program to return to your system's main command environment.

1.2.2 A Sample Program

The terminal session introduced six BASIC–PLUS commands: NEW, LIST, RUN, SAVE, OLD, and REPLACE. These and other commands are described in detail in Chapters 4 and 5 of this manual. Now let's look at the language itself, which is composed of statements.

The program used in the terminal session illustrates the syntax of the BASIC–PLUS language and contains several types of statements. Here again is the sample program:

```
10 PRINT "Enter a number between 1 and 15"
20 PRINT "and this program will calculate its factorial."
30 INPUT N
40 LET F=1
50 FOR X=1 TO N
60 LET F=F*X
70 NEXT X
80 PRINT "The factorial of";N;"is ";
85 PRINT NUM1$(F)
90 END
```

This program contains PRINT, INPUT, LET, FOR, NEXT, and END statements.

You must follow certain rules when you compose BASIC–PLUS statements; these rules are called the syntax of the BASIC–PLUS language. Look at line 10 of the sample program, which contains a PRINT statement. When you run the program, this line causes the system to print the following text on your terminal:

```
Enter a number between 1 and 15
```

In line 10, the word PRINT, the blank space after the word PRINT, and the quotation marks around the text are all part of the syntax of BASIC–PLUS. For example, one BASIC–PLUS syntax rule states that you must enclose text to be printed in quotation marks. Parts II, III, and IV of this manual describe the syntax rules of the BASIC–PLUS language in detail.

When you run the sample program, it:

- Prints information on the terminal to indicate the input it wants
- Accepts input
- Performs calculations on the input
- Prints the results, or output, on the terminal

Each of these operations corresponds to a statement or group of statements in the program:

- The PRINT statements in lines 10 and 20 print information that asks for input.
- The INPUT statement in line 30 accepts data.
- Lines 40 through 70 contain statements that perform calculations on the data. Line 40 is a LET statement, which assigns a value to a variable; lines 50 through 70 are a group of statements called a FOR loop, used to perform an operation repetitively.
- The PRINT statements in lines 80 and 85 print the results of these calculations on the terminal.

These and other BASIC–PLUS statements are described in detail in Parts II, III, and IV of this manual.

Chapter 2
BASIC–PLUS and RSTS/E

This chapter explains the relationship between the BASIC–PLUS command environment and the RSTS/E operating system. It also tells you how to enter and leave the BASIC–PLUS environment on different types of RSTS/E systems.

2.1 Terms

To understand how you, BASIC–PLUS, and RSTS/E work together, you need to know these terms:

Job

The unit that RSTS/E uses to keep track of you (and other users) during a terminal session. When you log in, the system creates a job for you and assigns it a job number. The system uses your job number to keep track of everything you do from the time you log in to the time you log out.

Run-Time System

System software that manages part of the RSTS/E system. For example, the BASIC–PLUS run-time system manages the BASIC–PLUS programming environment.

Keyboard Monitor

The part of a run-time system that you communicate with. When you work in the BASIC–PLUS programming environment, for example, you type commands to the BASIC–PLUS keyboard monitor. Each RSTS/E keyboard monitor has an identifying "prompt" it displays when it expects input from you. The BASIC–PLUS keyboard monitor's prompt is "Ready."

Default Keyboard Monitor

The main keyboard monitor that you work in on a RSTS/E system. You enter the default keyboard monitor after you log in. The system manager chooses the default keyboard monitor for a particular system.

Job Keyboard Monitor

The keyboard monitor that manages a job. Your job keyboard monitor is the same as the default keyboard monitor unless you change it. Changing your job keyboard monitor lets you work in a RSTS/E command environment that differs from the main command environment on your system.

You change your job keyboard monitor by using the SWITCH program. After you change your job keyboard monitor, you stay in that keyboard monitor until you log out or use SWITCH to change keyboard monitors again.

DCL (DIGITAL Command Language)

A set of commands available on many different DIGITAL systems. DCL commands perform basic tasks like copying files, printing files, and running programs. On RSTS/E, the DCL command environment is managed by the DCL run-time system. Like BASIC–PLUS, the DCL run-time system has a keyboard monitor.

CCL (Concise Command Language)

A shorthand way to call a RSTS/E system program, a DIGITAL–supplied program such as EDT, or a user program and give it one line of input on a single command line. The system manager chooses the CCL commands for a particular RSTS/E system.

2.2 The BASIC–PLUS Run-Time System

BASIC–PLUS is one of several RSTS/E run-time systems. You communicate with the BASIC–PLUS run-time system through a keyboard monitor that understands a set of commands. Many commands, such as NEW, OLD, COMPILE, and SAVE, act only on BASIC–PLUS programs. One command, RUN, can run both BASIC–PLUS programs and other types of programs:

- RSTS/E system programs
- DIGITAL–supplied programs such as the EDT editor
- User programs written in languages such as BASIC–PLUS–2, COBOL, and MACRO

Other keyboard monitor commands log you in and out (HELLO, BYE) and let you work with devices (ASSIGN, DEASSIGN). Appendix B of this manual summarizes all the BASIC–PLUS keyboard monitor commands.

Except for the summary in Appendix B, the rest of this manual discusses only the BASIC–PLUS keyboard monitor commands used for program development. Commands such as HELLO, BYE, ASSIGN, and DEASSIGN are described in the *RSTS/E System User's Guide.*

Besides keyboard monitor commands, you can use two other kinds of commands in the BASIC–PLUS environment: CCL commands, which your system manager defines, and DCL commands, available if your system has the DCL run-time system. To enter a CCL command, type its name; to enter a DCL command, type the letters "DCL", a space, and then the command name. Although CCL and DCL commands use other run-time systems, they return you to the BASIC–PLUS run-time system when they are finished. (Refer to the *RSTS/E System User's Guide* for information on CCL commands and the *RSTS/E DCL User's Guide* for information on DCL commands.)

Every run-time system has a name. The most common name for the BASIC–PLUS run-time system is "BASIC." It is possible, however, for a system manager to give the BASIC–PLUS run-time system a different name. In addition, some RSTS/E systems have more than one BASIC–PLUS run-time system. Each may have different optional features and a different name. Check with your system manager for information about BASIC–PLUS run-time systems on your system.

2.3 Types of RSTS/E Systems

Some RSTS/E systems have only the BASIC–PLUS run-time system; others have several run-time systems installed. On systems with several run-time systems, one run-time system is the default keyboard monitor; this default is often, but not always, BASIC–PLUS. You can recognize the BASIC–PLUS keyboard monitor by its "Ready" prompt on your terminal. Other RSTS/E systems have run-time systems such as DCL or RSX for the default keyboard monitor. You can recognize DCL by its "$" prompt and RSX by its ">" prompt. It is important for you to know your default keyboard monitor because it affects the way you work in the BASIC–PLUS environment.

2.4 Using BASIC–PLUS on a BASIC–PLUS System

If your system has only BASIC–PLUS or if BASIC–PLUS is your system's default keyboard monitor, you see the Ready prompt after you log in. You can use BASIC–PLUS commands such as NEW, OLD, RUN, and SAVE to work with BASIC–PLUS programs. You can also use CCL commands and run other types of programs available on your system. You stay in the BASIC–PLUS environment until you log out or until you use the SWITCH program to change your job keyboard monitor.

2.5 Using BASIC–PLUS on Other RSTS/E Systems

On RSTS/E systems that have DCL or another run-time system for the default keyboard monitor, you must enter the BASIC–PLUS environment before you can use BASIC–PLUS commands. As on BASIC–PLUS systems, you can use both BASIC–PLUS commands and CCL commands after you enter BASIC–PLUS, and you stay in BASIC–PLUS until you log out or use the SWITCH program. But how you enter BASIC–PLUS depends on what type of RSTS/E system you work on.

2.5.1 DCL Systems

On a DCL–based RSTS/E system, you can enter the BASIC–PLUS environment with the BASIC/BPLUS command or the SWITCH program. Both methods make BASIC–PLUS your job keyboard monitor. Enter the BASIC/BPLUS command in response to the "$" DCL prompt:

```
$ BASIC/BPLUS<RET>

Ready
```

To run the SWITCH program, enter:

```
$ RUN $SWITCH<RET>
Keyboard Monitor to switch to? BASIC<RET>

Ready
```

You can start using BASIC–PLUS commands when you see the Ready prompt. When you finish working in BASIC–PLUS, use the SWITCH program to return to the DCL environment or the BYE command to log out.

2.5.2 Other Non–BASIC–PLUS Systems

Some RSTS/E systems use RSX or other run-time systems for the default keyboard monitor. On these systems you enter BASIC–PLUS by running the SWITCH program.

The following example shows how to use SWITCH to enter the BASIC–PLUS environment on an RSX–based RSTS/E system:

```
>RUN $SWITCH<RET>
Keyboard Monitor to switch to? BASIC<RET>

Ready
```

You can start using BASIC–PLUS commands when you see the Ready prompt. When you finish working in BASIC–PLUS, use the SWITCH program to return to the default keyboard monitor or use the BYE command to log out.

Chapter 3
Overview of BASIC–PLUS Program Development

This chapter introduces the BASIC–PLUS commands and other features used to develop (write, run, edit, save, and debug) programs. It also defines terms used often in the command descriptions in Chapters 4, 5, and 6.

You enter BASIC–PLUS commands at a terminal, in response to the Ready prompt. Because commands are not part of a program, you do not use line numbers. Commands cannot be abbreviated except where noted. When you finish typing a command, press the RETURN key. If the command is correct, the system executes it and displays the Ready prompt when it is finished. If the command is incorrect, the system displays an error message followed by the Ready prompt.

3.1 Source and Translated Programs

There are two types of BASIC–PLUS programs: source programs and translated programs. Both source and translated programs can be stored on disk. A source program:

- Is stored on disk in human-readable (ASCII) format
- Must be translated by BASIC–PLUS before it can be executed
- Is available to you for listing (displaying at the terminal) and for changing (editing and debugging)

Source programs are stored with .BAS file types and are sometimes called BAS programs. Most of the commands and operations described in Part I work on source programs.

A translated program:

- Is not human-readable. Do not display it on your terminal or print it on the line printer.
- Has already been translated by BASIC–PLUS and is stored in its translated form, called intermediate code.
- Is available to you for running, but not for listing or changing.

Translated programs are stored with .BAC file types and are sometimes called BAC programs.

3.2 The Current Program

The current program is the program that you are currently working with in memory. You can be writing, editing, listing, running, or debugging it. This "current program" can be a new program that you created during the present timesharing session or an existing ("old") program that you retrieved from your disk storage area. It can be a source program or a translated program.

When your current program is a source program, BASIC–PLUS always keeps a translated copy of it available so you can run it. But when your current program is a translated program, you can only run it. BASIC–PLUS does not keep a source copy available for listing or changing.

Your area in memory can contain only one current program at a time. Creating, retrieving, or running another program (a BASIC–PLUS program or other type of program) erases your current program from memory.

3.3 Command and Statement Summary

Table 3–1 summarizes the BASIC–PLUS commands and shows where in this manual they are described. Tables 3–2 and 3–3 list BASIC–PLUS statements and RSTS/E features that are helpful in developing programs.

Table 3–1: BASIC–PLUS Commands

Command	Purpose
APPEND	Merges the contents of a previously saved source program with the current program. See page 5–8.
CATALOG or CAT	Displays your file directory. See page 4–9.
CCONT	Continues execution of the current program after a STOP; also detaches your job from the terminal. CCONT is used in debugging and requires privilege. See page 6–9.
COMPILE	Saves a translated program in a disk file. See page 4–8.
CONT	Continues execution of the current program after a STOP. CONT is used in debugging. See page 6–9.

(continued on next page)

Table 3–1: BASIC–PLUS Commands (Cont.)

Command	Purpose
DELETE	Removes one or more lines from your current program. See page 5–2.
EXTEND	Puts BASIC–PLUS in EXTEND mode; allows you to run programs with EXTEND mode features. See page 4–12.
LENGTH	Displays the amount of memory used by the current program and the maximum amount it can use. See page 4–10.
LIST	Displays the current program on your terminal. See page 4–5.
NEW	Clears memory and names a new program. See page 4–1.
NOEXTEND	Puts BASIC–PLUS in NOEXTEND mode. You can no longer run programs with EXTEND mode features. See page 4–12.
OLD	Retrieves a saved source program from disk and places it in memory. See page 4–4.
RENAME	Changes the name of the program currently in memory. See page 5–3.
REPLACE	Copies the source program currently in memory into a disk file; replaces any existing file with the same name. See page 5–6.
RUN	Runs a program. See page 4–6.
SAVE	Copies the source program currently in memory into a disk file. SAVE does not replace an existing file with the same name. See page 4–7.
SCALE	Displays or sets the scale factor. See page 4–11.
UNSAVE	Deletes a file from a directory. See page 5–6.

Table 3–2: BASIC–PLUS Statements

Statement	Purpose
GOTO	Used in immediate mode to continue program execution at a specified line after a STOP. Immediate mode GOTO statements are useful in debugging. See page 6–9.
KILL	Deletes a file from a directory. See page 5–7.
NAME–AS	Changes a file name, file type, or protection code. See page 5–4.
PRINT	Displays values of program variables that you specify. Use immediate mode PRINT statements in debugging to display current variable values after a STOP. See page 6–11.
PRINT LINE	Displays the BASIC–PLUS variable LINE. When you halt a program with CTRL/C, LINE contains the line number where the halt occurred. See page 6–12.
STOP	Halts a program's execution at a specified line. STOP is used mainly in debugging. See page 6–8.

Table 3–3: Control Characters and Terminal Keys

Key	Purpose
CTRL/C	Halts execution of the current program. See page 6–12.
CTRL/O	Stops and restarts terminal output while a program is running. CTRL/O can be useful in debugging. See page 6–13.
CTRL/S	Suspends terminal output while a program is running. CTRL/S can be useful in debugging. See page 6–13.
CTRL/Q	Resumes terminal output suspended by CTRL/S while a program is running. CTRL/Q can be useful in debugging. See page 6–13.
CTRL/U	Deletes the current terminal line. CTRL/U is useful in program editing. See page 5–2.
DELETE key	Erases the last character typed. The DELETE key is useful in program editing. See page 5–2.
RUBOUT key	Appears on some terminal keyboards instead of the DELETE key. Like the DELETE key, the RUBOUT key erases the last character typed and is useful in program editing. See page 5–2.

Chapter 4
Creating and Running a BASIC–PLUS Program

This chapter describes the BASIC–PLUS commands that create, retrieve, display, run, and save programs: NEW, OLD, LIST, RUN, SAVE, and COMPILE. In addition, it describes commands for getting information about programs (CATALOG, LENGTH, SCALE) and commands used to change modes (EXTEND, NOEXTEND).

The commands in this chapter are grouped by function; they are not necessarily in the order that you would use them when developing a BASIC–PLUS program. If you are a new BASIC–PLUS user, review the sample terminal session in Chapter 1 of this manual before reading this chapter. The terminal session shows you the steps to follow in developing a simple BASIC–PLUS program.

4.1 Creating a Program

To create a new BASIC–PLUS program, use the NEW command and then enter the program statements into the system.

4.1.1 The NEW Command

The NEW command clears memory and names a new program. NEW deletes any program currently in memory. The command format is:

NEW [filename]

If you do not specify a file name, NEW prompts you for one. If you press the RETURN key in response to the prompt, NEW names the program NONAME. The following examples show different ways to enter the NEW command:

```
NEW<RET>
New File name--CASH01<RET>
```
NEW creates a program named CASH01.

```
NEW CASH01<RET>
```
NEW creates a program named CASH01.

```
NEW<RET>
New File name--<RET>
```
NEW creates a program named NONAME.

4.1.2 Entering the Program

After you create a program in memory with the NEW command, you enter the BASIC–PLUS program statements. To enter a BASIC–PLUS statement, type a line number, a space or tab, and the statement. Then press the RETURN key. Follow these steps for each program statement. For example:

```
10 PRINT "B"<RET>
20 X=2*3<RET>
30 PRINT X<RET>
40 END<RET>
```

4.1.2.1 Correcting Typing Errors

If you make a typing mistake, correct it with the DELETE key (RUBOUT on some terminals) or CTRL/U. The DELETE key erases individual characters. On video terminals, the deleted characters disappear; on hardcopy terminals, the deleted characters are printed between backslashes. Retype the correct characters on the same line. (Because a hardcopy terminal prints deleted characters between backslashes, a line with deletions can become difficult to read. You can reprint the line by pressing CTRL/R. Hold the CTRL key down and press the R key.)

CTRL/U deletes the current line on your terminal. To enter a CTRL/U, hold the CTRL key down and press the U key. Unlike the DELETE key, CTRL/U does not erase characters from the screen. Instead, the system displays "^U" and moves the cursor to the next line. You can then retype the line.

You can use DELETE or CTRL/U only before you press the RETURN key. If you notice a typing mistake after you press the RETURN key, retype the line number followed by the correct statement.

See the following chapters or manuals for more information:

BASIC–PLUS editing features	Chapter 5
Using a terminal	*RSTS/E System User's Guide*

4.1.2.2 Correcting Syntax Errors

BASIC–PLUS checks each statement you enter for syntax errors after you press the RETURN key. If the statement is correct, the cursor waits on the next line for your input. If the statement has one or more syntax errors, BASIC–PLUS prints an error message below the incorrect statement, followed by the Ready prompt. To correct a syntax error, retype the line number followed by the corrected statement. For example:

```
10 PRINT "THIS IS A TEST"
20 PRINT THIS IS A TEST
? Syntax error at line 20

Ready

20 PRINT "THIS IS A TEST"
```

4.1.2.3 Program Translation

Each time you enter a correct line of source code, BASIC–PLUS translates the line into code that the system can execute. BASIC–PLUS keeps this translated code in your current program's memory area.

Because BASIC–PLUS translates each line of your program as you enter it, you can run your program as soon as you finish entering it. See the RUN command for more information.

4.1.2.4 The TEMPnn.TMP File

When you create a program with the NEW command, the system creates the file TEMPnn.TMP in your account on the public structure; nn is your job number. TEMPnn.TMP contains the text of your source program and any changes you make to that text. Do not work with the TEMPnn.TMP file or type it on your terminal. The system uses TEMPnn.TMP as a "scratch file" and deletes it when you log out. It is noted here because it uses disk space in your area and appears in a directory listing of your files during a BASIC–PLUS session.

NOTE

Besides using NEW and entering a program in BASIC–PLUS, you can also create (and edit) a BASIC–PLUS source program with a text editor such as EDT. Assign a .BAS file type when you name the file. This method is not as interactive as using the BASIC–PLUS editing features; for example, you may not discover syntax errors in your program until you retrieve it with the OLD command or run it with the RUN command. However, you can take advantage of text editing features not available in BASIC–PLUS.

4.2 Calling an Existing Program – The OLD Command

The OLD command retrieves a previously saved BASIC–PLUS source program and places it in memory. OLD deletes the program currently in memory.

As OLD retrieves each line of the saved source program, it:

- Translates the source code into executable code
- Places the translated code in your current program's memory area

Besides translating the source program so you can run it (or save it on disk with the COMPILE command), the OLD command also makes the source program available to you for listing or changing. You must retrieve a saved source program with OLD before you can list or edit the program.

OLD has the format:

OLD [filespec]

where filespec is a complete RSTS/E file specification of the form:

dev:[acct]filename.typ/switch(es)

Except for the file name, all parts of the file specification are optional. If you type OLD and press the RETURN key, you are prompted to enter a file name:

```
OLD RET
OLD file name--
```

Press the RETURN key to retrieve NONAME.BAS; otherwise, enter all or part of a file specification. The OLD command default is a .BAS file in your account on the public structure. You receive an error message if the file does not exist or is protected against you.

The following examples show how OLD works:

`OLD TAXES`	Retrieves the file TAXES.BAS from your account on the public structure. (You must specify the file type if it is not .BAS.)
`OLD TAXES.TTL`	Retrieves TAXES.TTL from your account on the public structure. When you specify a file type other than .BAS, be sure it is a BASIC–PLUS source program.
`OLD DM1:[200,233]TAXES`	Retrieves TAXES.BAS from [200,233] on DM1:.

Usage Notes

1. Make sure that any file you specify in the OLD command is a BASIC–PLUS source program. If you specify a different kind of file, BASIC–PLUS prints numerous error messages on your terminal; use CTRL/C to abort the OLD command.

2. The system creates a TEMPnn.TMP file in your area when you retrieve a saved source program with the OLD command. The file contains the text of the source program you just retrieved. See the NEW command for information about this file.

4.3 Displaying a Program – The LIST Command

After you enter a new program, retrieve an existing program, or edit a program, display it on your terminal with the LIST command to make sure all statements are correct. You can display the entire program or one or more lines; you can display the program with or without header information. When you ask for header information, LIST displays the program name and the current date and time. You can use LIST to display only the program currently in memory.

The LIST command has the format:

LIST[NH] [line numbers]

NH means no header. The following examples show how the LIST command works:

`LIST`	Displays the entire program including header information.
`LISTNH`	Displays the entire program without header information.
`LIST 100`	Displays line 100 with header information.
`LIST 100-200`	Displays lines 100 through 200 with header information.
`LISTNH 25,34,60-85`	Displays lines 25, 34, and 60 through 85 without header information.

You need not specify lines in sequential order; lines are displayed in the order you request.

LIST prints a ? at the left of a program line that contains a syntax error. For example:

```
LISTNH
10 LET A=25
15 LET B=38
?20 PPRINT A+B

Ready
```

Note that LIST sends output to your terminal only. Use the QUE program, the DCL PRINT command, or SAVE LP: to print a copy of your program on the line printer. The QUE program and the DCL PRINT command are the recommended methods; they place your request in the printer queue. SAVE LP:, on the other hand, sends output directly to the line printer if it is free. See the following manuals or sections in this manual for more information:

SAVE LP:	Section 4.5
QUE program	*RSTS/E System User's Guide*
DCL PRINT command	*RSTS/E DCL User's Guide*

4.4 Running a Program – The RUN Command

The RUN command runs (executes) a BASIC–PLUS source program, a BASIC–PLUS translated program, or any other executable program (such as the executable form of a BASIC–PLUS–2 or COBOL program). You can run the current program or a program stored on disk.

The RUN command has the format:

RUN[NH] [filespec]

NH means no header. Before executing the current program, the RUN command prints a program header that consists of the program name and the current system date and time. When you run the current program, you can add NH to the RUN command to avoid printing this header information.

The filespec is a complete RSTS/E file specification of the form:

dev:[acct]filename.typ/switch(es)

Use the file specification to run a program not currently in memory. When you run a program not currently in memory, your current program is deleted from memory.

The following examples show how RUN works:

`RUN`	Runs the current program.
`RUNNH`	Runs the current program without printing header information.
`RUN TAXES`	Runs the program TAXES in your account on the public structure.
`RUN DM1:[200,230]TAXES`	Runs the program TAXES in [200,230] on DM1:.

When you specify the file name of a program not currently in memory:

1. The system looks first for an executable program with the name you specify. An executable program is a file whose protection code includes the value 64. A BASIC–PLUS translated program (.BAC file) is an executable program. If an executable program exists, the system loads it and runs it.

2. If no executable program exists, the system looks for a BASIC–PLUS source program (.BAS file) with the name you specify. If the source program exists, the system loads it, translates it, and runs it.

Usage Notes

1. If you have programs with duplicate file names in your directory, always specify the .BAC or .BAS file type in the RUN command. If you do not, it is possible for the system to run an executable file that is not a BASIC–PLUS program.
2. In NOEXTEND mode, you cannot run a saved program whose file name begins with the letters NH. BASIC–PLUS interprets these letters as "no header" instead of running the specified program.

4.5 Saving a Source Program – The SAVE Command

The SAVE command copies the source program currently in memory into a disk file.

SAVE has the format:

SAVE [filespec]

where filespec is a complete RSTS/E file specification of the form:

dev:[acct]filename.typ/switch(es)

All parts of the file specification are optional.

The following examples show how SAVE works:

`SAVE`	Creates a file in your account on the public structure. The file name is your current program name; the file type is .BAS.
`SAVE NEWNAM`	Creates a file NEWNAM.BAS in your account on the public structure.
`SAVE NEWNAM.E12`	Creates a file NEWNAM.E12 in your account on the public structure.
`SAVE DL1:[3,222]`	Creates a file in [3,222] on DL1:. The file name is your current program name; the file type is .BAS.

SAVE does not supersede existing files. If the file you specify already exists, you receive the message:

```
?File exists-RENAME/REPLACE
```

Reenter the SAVE command and specify a new file name or use the REPLACE command, described in Section 5.3, to replace the existing file with your current program.

Usage Notes

1. You can use SAVE LP: to print your current program on the line printer. However, it is recommended that you use the QUE program or the DCL PRINT command instead of SAVE LP:. The QUE program and the DCL PRINT command place your request in the line printer queue. Your request is printed when the printer is available and contains header pages to identify it. Use SAVE LP: if your system does not have the QUE program or the DCL PRINT command. SAVE LP: does not queue your request or print identifying header pages; it simply prints the program on the line printer if it is free. The system displays a message if the line printer is busy, and your program does not get printed. See the *RSTS/E System User's Guide* for information about the QUE program and the *RSTS/E DCL User's Guide* for information about the DCL PRINT command.

2. If you use NOEXTEND mode, do not save a program with a file name that begins with the letters NH. When you specify the program in the RUN command, BASIC–PLUS interprets NH as "no header" instead of running the specified program.

4.6 Saving a Translated Program – The COMPILE Command

There are two types of BASIC–PLUS code: source code and translated code. When you use the RUN command to run a program, the system executes translated code, not source code. If only source code has been saved, BASIC–PLUS automatically translates the source code before it is executed.

Translating a source program can take considerable system time. You can save this time by using the COMPILE command, which saves an image of a translated BASIC–PLUS program in a disk file.

You can use the COMPILE command only on a program that is currently in memory. Thus, you must retrieve an existing source program with the OLD command before using COMPILE. (OLD translates the program as it places it in memory.) The COMPILE command has no effect on either the current program or its source file on disk.

The COMPILE command has the format:

```
COMPILE [filespec]
```

where filespec is a complete RSTS/E file specification of the form:

```
dev:[acct]filename.typ/switch(es)
```

All parts of the file specification are optional. The default file name is your current program name; the default file type is .BAC. A translated program can be stored only on disk; therefore, the dev: portion of the file specification must be a disk pack.

The following examples show how the COMPILE command works:

`COMPILE`	Saves a translated image of your current program in a .BAC file in your account on the public structure. The file name is the current program name.
`COMPILE PROG1`	Saves a translated image of your current program in the file PROG1.BAC in your account on the public structure.
`COMPILE DL1:`	Saves a translated image of your current program on disk pack DL1:. The file name is the current program name; the file type is .BAC.

Usage Notes

1. Translated programs are stored with a default protection code that is the sum of the system default (usually 60) plus 64, which indicates an executable file. To store the translated program with a different protection code, use the /PROTECT switch. For example:

   ```
   COMPILE/PR:40
   ```

   The translated program is saved with a protection code of 104, the sum of 40 and 64.

   As in previous versions of RSTS/E, you can also enclose the protection code in angle brackets. However, it is recommended that you use the /PROTECT switch. See the *RSTS/E System User's Guide* for more information about protection codes.

2. Each translated file requires a minimum of 7 disk blocks. Thus, while storing translated programs saves execution time, translated programs can use more disk space than source programs. The translated file is contiguous if enough disk space is available. Otherwise, the system creates a noncontiguous file.

3. If you use NOEXTEND mode, do not save a translated program with a file name that begins with the letters NH. When you specify the program in the RUN command, BASIC–PLUS interprets the letters as "no header" instead of running the specified program.

4.7 Getting Information – CATALOG, LENGTH, and SCALE

This section describes how to get a directory listing of files, display the length of the current program, and display the scale factor.

4.7.1 Listing Files in a Directory – The CATALOG Command

The CATALOG command displays information about all files in your directory on the public structure or a private disk. For each file, CATALOG lists the file name and file type, file size in blocks, protection code, date of creation, and date and time of last access. CATALOG has no effect on your current program.

You can also specify a directory other than your own. If you do, CATALOG displays information about files in the directory to which you have read access. (Your system manager can change the system so that CATALOG displays information on all files in the directory you specify, not just files to which you have read access.)

The CATALOG command has the format:

CATALOG [dev:[acct]]

The following examples show how CATALOG works. You can abbreviate the command to CAT. The examples assume you have read access to the files.

`CATALOG`	Lists all files in your account on the public structure.
`CATALOG [100,102]`	Lists all files in [100,102] on the public structure.
`CATALOG [1,2]`	Lists all files in the system library account [1,2].
`CATALOG DM0:`	Lists all files in your account on disk DM0:.
`CATALOG MM0:[200,222]`	Lists all files in account [200,222] on magnetic tape unit MM0:.

Here is a sample directory listing produced by the CATALOG command:

```
CAT
AVERAGE.BAS     2       60      28-Oct-81 28-Oct-81 02:37 PM
PERCNT .BAS     2       60      28-Oct-81 28-Oct-81 03:22 PM
TERM   .DOC     11      60      28-Oct-81 28-Oct-81 04:10 PM
EXPENS .BAS     29      60      28-Oct-81 28-Oct-81 04:40 PM
```

Usage Note

You can also use the DCL DIRECTORY command or the PIP or DIRECT programs to get a directory listing of your files. Unlike the CATALOG command, however, these methods delete your current program from memory.

4.7.2 Displaying a Program's Length in Memory – The LENGTH Command

The LENGTH command displays the current program's length in memory and the maximum amount of memory that the program can use. The length of the current program is displayed in K words to the next highest 1K increment. (K stands for 1024.)

The LENGTH command has the format:

LENGTH

For example:

```
LENGTH<RET>
5(16)K of memory used
```

In this example, the current program uses 5K words of memory; its maximum allowed length is 16K words. The minimum length displayed for a current program is 2K words, even when you have no program in memory. The system reserves this memory space for its own use.

4.7.3 Displaying the SCALE Factor – The SCALE Command

The SCALE command controls the BASIC–PLUS scaled arithmetic feature. This feature, available only on systems with the double-precision math package, lets you avoid accumulated roundoff and truncation errors in fractional compututations.

You work with scaled arithmetic by setting the scale factor. The scale factor determines the number of decimal places used in floating-point calculations during program execution. However, BASIC–PLUS establishes the scale factor for a program during translation, not during execution.

The SCALE command has two functions: it displays and sets the scale factor. This section shows how to use SCALE to display the scale factor. See Section 11.10 for a complete description of how to work with scaled arithmetic and how to use the SCALE command to set the scale factor.

The scale factor has a default value of 0, which means that BASIC–PLUS uses no scale factor when translating a program. On systems with the single-precision math package, the scale factor is always 0. (On systems with the double-precision math package, the system manager can change the scale factor's default value.)

BASIC–PLUS keeps track of two scale factors: the current scale factor, used when your current program was translated, and the pending scale factor, to be used the next time translation occurs. One value is printed if both scale factors are the same.

To display the scale factor, type:

```
SCALE
```

For example:

```
SCALE RET
6 ,2
```

The first number is the pending scale factor; the second is the current scale factor. Thus, this display tells you that your current program was translated using a scale factor of 2 and that next time translation occurs the scale factor will be 6.

4.8 Changing Modes – EXTEND and NOEXTEND

BASIC–PLUS has two modes of operation: EXTEND mode and NOEXTEND mode. When you work in EXTEND mode, you can use features not available in NOEXTEND mode, such as long variable names. However, the format requirements for your program are more stringent in EXTEND mode than in NOEXTEND mode.

EXTEND is the default mode on most systems; however, the system manager can change the default mode to NOEXTEND. It is recommended that you write programs in EXTEND mode for compatibility with BASIC–PLUS–2. NOEXTEND mode, on the other hand, is compatible with previous versions of BASIC–PLUS. The examples in this manual are written to execute in EXTEND mode.

EXTEND and NOEXTEND can be used either as commands or as program statements. Their actions differ depending on how you use them. This section shows you how EXTEND and NOEXTEND work when used as commands. Section 7.5 compares the actions of EXTEND and NOEXTEND when used as commands and program statements and explains the difference between EXTEND and NOEXTEND program formats.

The following examples assume you are logged into a system where NOEXTEND is the default mode. To change to EXTEND mode, type:

```
EXTEND
```

You stay in EXTEND mode until you either:

- Enter the NOEXTEND command
- Use the NOEXTEND statement in your current program
- Run an executable (but not a source) program
- Enter a CCL or DCL command
- Switch to another keyboard monitor
- Log out

To change back to NOEXTEND mode, type either:

```
NOEXTEND
```

```
NO EXTEND
```

No change occurs if you enter the command for the mode that the system is currently in.

Chapter 5
Modifying BASIC–PLUS Programs

This chapter describes commands and features for modifying existing BASIC–PLUS programs. Topics include:

- Editing a program
- Changing a program's name or file specification
- Replacing or deleting a program
- Merging programs in memory

5.1 Editing a BASIC–PLUS Program

If you read Chapters 1 and 4 of this manual, you already know how to edit a BASIC–PLUS program. These chapters showed you how to correct program statements with the DELETE key and CTRL/U and how to replace an incorrect statement with a new statement. This section reviews these topics and introduces the DELETE command.

The editing features of BASIC–PLUS can be used only on the program that is currently in memory. The program must be a source program; you cannot edit a translated program. Thus, before editing a BASIC–PLUS program, you must retrieve its source file from disk storage with the OLD command. Use the LIST command to display the program after you retrieve it. OLD and LIST are discussed in Chapter 4.

5.1.1 Entering New Program Statements

After retrieving a program, you add new statements or replace existing statements the same way you do when working with a new program. To add a new statement, type a new line number, a space or tab, and the

statement. Then press the RETURN key. BASIC–PLUS informs you of syntax errors after you press the RETURN key. To replace a program statement, simply retype the statement using the same line number. You need not enter line numbers in order; BASIC–PLUS orders the line numbers as you enter them. See Chapter 1 or 4 for examples of entering statements.

5.1.2 Using the DELETE Key and CTRL/U

Use the DELETE key (RUBOUT on some terminals) and CTRL/U to correct typing mistakes as you enter a program statement, but before you press the RETURN key. The DELETE key erases the last character you typed; CTRL/U erases the entire line and moves the cursor to the next line. The following examples show the actions of the DELETE key and CTRL/U on a video terminal.

Enter the following statement on your terminal but do not press the RETURN key:

```
10 PRINT This
```

Note that you omitted the first quotation mark. Press the DELETE key four times. The DELETE key erases the characters "This" and moves the cursor back four spaces:

```
10 PRINT 
```

Now use CTRL/U to erase the rest of the line. The system displays "^U" and moves the cursor to the beginning of the next line on the screen:

```
10 PRINT ^U
```

You can now enter another program statement.

Usage Notes

1. On hardcopy terminals, the DELETE or RUBOUT key echoes the erased characters between backslashes. If you use this key several times, the line may become difficult to read. Use CTRL/R to reprint the line before you press the RETURN key.
2. CTRL/U erases the current terminal line. If you continue a BASIC–PLUS statement on more than one line, the current terminal line may contain only part of the statement. CTRL/U erases only the portion of the statement on the current terminal line.

5.1.3 The DELETE Command

The DELETE command erases one or more lines from the current program. DELETE has the format:

DELETE [line number(s)]

You can delete a single line by typing just the line number followed by the RETURN key. When you delete more than one line, you need not specify line numbers in sequential order. Typing DELETE without a line number deletes all lines from your current program.

The following examples show how DELETE works:

`DELETE`	Deletes all lines from the current program.
`DELETE 10`	Deletes line 10 from the current program.
`10`	Deletes line 10 from the current program.
`DELETE 100-200`	Deletes lines 100 through 200 from the current program.
`DELETE 100-200,24,35,300-400`	Deletes lines 100 through 200, line 24, line 35, and lines 300 through 400 from the current program.

Usage Notes

1. Be sure to specify one or more line numbers in the DELETE command unless you want to delete all lines in your current program.
2. Before deleting a program line, check your program for references to that line number (in GOTO statements, for example) and make the necessary changes to the program.
3. The BASIC–PLUS DELETE command differs from the DCL DELETE command. The DCL DELETE command deletes a file from a directory.

NOTE

You can also use a text editor such as EDT to edit any BASIC–PLUS source program that you have saved in a disk file. While this method does not detect syntax errors, it lets you take advantage of text editing features not available in BASIC–PLUS.

5.2 Changing a Program's Name or File Specification

The RENAME command and the NAME–AS statement both change the name of a program. However, they perform different functions. The RENAME command changes the name of the current program; the NAME–AS statement can change a program's file name, file type, and protection code.

5.2.1 The RENAME Command

The RENAME command changes the name of the program currently in memory. The old program name is discarded. The RENAME command does not change the file name of a saved program on disk. The command has the format:

RENAME filename

where filename is a RSTS/E file name.

For example, to change the name of your current program to NEWNAM, type:

```
RENAME NEWNAM
```

If you now use SAVE to save your program, it is saved in the file NEWNAM.BAS by default.

Usage Note

The BASIC–PLUS RENAME command differs from the DCL RENAME command. The DCL RENAME command renames a file.

5.2.2 The NAME–AS Statement

Unlike the RENAME command, which changes the name of the program currently in memory, the NAME–AS statement can change the file name, file type, and protection code of a program stored on disk.

NAME–AS can be used in a BASIC–PLUS program or as an immediate mode statement. When used as an immediate mode statement (without a line number), its acts like a command; that is, it is executed as soon as you press the RETURN key. This section discusses its use as an immediate mode statement. See Chapter 14 for information on using NAME–AS in a BASIC–PLUS program.

The NAME–AS statement has the format:

NAME <string> AS <string>

where string is all or part of a RSTS/E file specification of the form:

"dev:[acct]filename.typ/PR[OTECT]:n"

Replace n with a valid RSTS/E protection code.

As in previous versions of RSTS/E, you can also enclose the protection code in angle brackets. However, it is recommended that you use the /PROTECT switch. See the *RSTS/E System User's Guide* for more information about protection codes.

You must use single or double quotation marks around both strings. The file specified in the first string is renamed to the file specified in the second string.

NAME–AS defaults to your account on the public structure. When you specify a file located in another device or account, you need only specify the device or account in the first string. However, to retain a file type, you must specify it in both strings, even if you do not want to change it. If you do not, the renamed file has no file type.

The following examples show how NAME–AS works:

```
NAME "OLD.BAS" AS "NEW.BAS"
```

Renames the file OLD.BAS to NEW.BAS.

```
NAME "OLD.BAS" AS "NEW"
```

Renames the file OLD.BAS to NEW. The file now has no file type.

```
NAME "DM0:OLD.BAS" AS "DM0:NEW.BAS"
```

Renames the file stored in your account on DM0: from OLD.BAS to NEW.BAS.

```
NAME "DM0:OLD.BAS" AS "NEW.BAS"
```

Performs the same action as the previous example. It renames the file stored in your account on DM0: from OLD.BAS to NEW.BAS.

```
NAME "[200,201]OLD.BAS" AS "[200,201]NEW.BAS"
```

Renames the file stored in [200,201] on the public structure from OLD.BAS to NEW.BAS.

```
NAME "[200,201]OLD.BAS" AS "NEW.BAS"
```

Performs the same action as the previous example. It renames the file stored in [200,201] on the public structure from OLD.BAS to NEW.BAS.

```
NAME "OLD.BAS" AS "OLD.BAS/PR:40"
```

Changes the protection code of OLD.BAS from its present value to 40.

```
NAME "OLD.BAS" AS "NEW.BAS<40>"
```

Renames the file OLD.BAS to NEW.BAS and changes its protection code from its present value to 40.

Usage Notes

1. NAME–AS cannot copy files. Thus, while NAME–AS can change the file name, file type, or protection code of a file located in another account or device, NAME–AS cannot transfer a file to a different account or device.

2. To use NAME–AS to change a file name or protection code, you must have write access to the file.

3. When you use NAME–AS to assign protection codes, the results depend on your privilege.

   Only privileged users can assign a protection code greater than 63 to a source or data file or a protection code less than 64 to an executable file. If a nonprivileged user specifies a protection code greater than 63 for a source or data file, the system subtracts 64 (the executable code) from the value specified. For example:

   ```
   NAME "NEW.BAS" AS "NEW.BAS/PR:104"
   ```

   The system assigns a protection code of 40.

If a nonprivileged user specifies a protection code less than 64 for an executable file, the system adds 64 to the value specified. For example:

```
NAME "NEW.BAC" AS "NEW.BAC/PR:40"
```

The system assigns a protection code of 104, the sum of 64 and 40.

5.3 Replacing a Saved Program – The REPLACE Command

The REPLACE command replaces a previously saved BASIC–PLUS source program with the program currently in memory. REPLACE performs basically the same function as SAVE, but, unlike SAVE, it destroys any existing source program on the disk with the same name.

The REPLACE command has the format:

REPLACE [filespec]

where filespec is a RSTS/E file specification of the form:

dev:[acct]filename.typ/switch(es)

All parts of the file specification are optional. The following examples show how REPLACE works:

`REPLACE`	Copies the current program to a file in your account on the public structure. The file name is your current program name; the file type is .BAS. Replaces any file with the same file name and type.
`REPLACE NEWNAM`	Copies the current program to the file NEWNAM.BAS in your account on the public structure. Replaces any existing NEWNAM.BAS file.
`REPLACE DM0:[12,24]NEWNAM.V1`	Copies the current program to the file NEWNAM.V1 in [12,24] on DM0:. Replaces any existing file with the same file specification.

5.4 Deleting a Saved Program

You can delete a source or translated program from disk storage with the UNSAVE command or the KILL statement.

5.4.1 The UNSAVE Command

The UNSAVE command deletes a file from a directory. The file can be a BASIC–PLUS program or other type of file.

The command has the format:

UNSAVE [filespec]

where filespec is a RSTS/E file specification of the form:

dev:[acct]filename.typ/switch(es)

All parts of the file specification are optional. The default file name is your current program name; the default file type is .BAS. You must have write access to the file. The following examples show how UNSAVE works:

`UNSAVE`	Deletes from your account on the public structure a .BAS file with the same name as the current program.
`UNSAVE TAXES`	Deletes the file TAXES.BAS from your account on the public structure.
`UNSAVE TAXES.BAC`	Deletes the file TAXES.BAC from your account on the public structure.
`UNSAVE DL1:`	Deletes from your account on DL1: a file with the same name as the current program and a .BAS file type.
`UNSAVE [200,201]TAXES.BAC`	Deletes the file TAXES.BAC from [200,201] on the public structure.

5.4.2 The KILL Statement

Like the UNSAVE command, the KILL statement deletes a file from a directory. The file can be a BASIC–PLUS program or other type of file.

The KILL statement can be used in a BASIC–PLUS program or as an immediate mode statement. Its use as an immediate mode statement is discussed here; see Chapter 14 for information on using the KILL statement in a program.

The KILL statement has the format:

KILL <string>

where <string> is a RSTS/E file specification of the form:

"[dev:[acct]]filename.typ"

You must enclose the file specification in quotation marks. The KILL statement defaults to a file in your account on the public structure, but, unlike the UNSAVE command, it has no default file name or type. You must specify the file name and type. You must also have write access to the file.

The following examples show how the KILL statement works:

`KILL "TAXES.BAS"`	Deletes the file TAXES.BAS from your account on the public structure.
`KILL "DL1:TEST.BAC"`	Deletes the file TEST.BAC from your account on DL1:.
`KILL "DL1:[2,124]PROG1.BAS"`	Deletes the file PROG1.BAS from account [2,124] on DL1:.

5.5 Merging Programs – The APPEND Command

The APPEND command merges the contents of a previously saved BASIC–PLUS source program with the program currently in memory. The line numbers in the two programs determine how they are combined (see the Usage Notes).

The command has the format:

APPEND [filespec]

where filespec is a RSTS/E file specification of the form:

dev:[acct]filename.typ/switch(es)

If you do not specify a file name, APPEND prompts you for one. The file you specify must be a BASIC–PLUS source program. Note that APPEND merges a copy of the saved program with your current program; the disk file is not affected.

The following examples show how APPEND works:

`APPEND` RET `Old file name--` RET	Merges the file NONAME.BAS in your account on the public structure with your current program.
`APPEND` RET `Old file name--PROG2`	Merges the file PROG2.BAS in your account on the public structure with your current program.
`APPEND PROG2`	Merges the file PROG2.BAS in your account on the public structure with your current program.
`APPEND DM0:[12,12]PROG.V2`	Merges the file PROG.V2 in account [12,12] on DM0: with your current program.

Usage Notes

1. APPEND works on source programs only, not translated programs. Both the program currently in memory and the program you specify in the APPEND command must be source programs.
2. APPEND combines the two programs according to their line numbers. The combined program contains the lines from both programs in ascending order. If both programs contain the same line number, the line from the appended program replaces the line in memory.
3. To save the new program, use the SAVE or REPLACE command.
4. APPEND lets you work with modular programs. You can write small programs that perform functions you need in several programs and store them in separate source files. You can then use APPEND to combine in memory the specific set of modules you want to run.
5. You can include immediate mode statements (that is, statements with no line numbers) in a source file to be appended to your current program. (Immediate mode is described in Chapter 6.)

BASIC–PLUS executes immediate mode statements when it encounters them during an APPEND. They do not become part of your current program in memory; however, they can be used to modify variables in the program.

The following example shows a possible use of immediate mode statements in a source file. The first statement in the file is an immediate mode PRINT statement, which is executed during the APPEND. (The rest of the statements in the file, which have line numbers, become part of the current program.) The PRINT statement tells you which module of a large program is being appended to the current program and also prints the time of day.

```
PRINT "Appending Payroll Calculation Routine, ";TIME$(0%)
20010
20020
 .
 .
 .
32767 END
```

Note that your system manager can change BASIC–PLUS so that it does not execute immediate mode statements during an APPEND.

6. The BASIC–PLUS APPEND command differs from the DCL APPEND command. The DCL APPEND command appends one file to the end of another.

   You should not use the DCL APPEND command to merge BASIC–PLUS source files. When you later retrieve or run the resulting source file, you lose program lines if the file contains more than one END statement. BASIC–PLUS only processes statements that appear before the first END statement.

   Do not merge .BAC files with the DCL APPEND command. You will not be able to execute the resulting file.

Chapter 6
Immediate Mode and Program Debugging

A program rarely works correctly the first time you run it. It must be tested and corrected. This process is called debugging.

This chapter describes the features that BASIC–PLUS and RSTS/E provide for debugging BASIC–PLUS programs. The principal BASIC–PLUS debugging feature is immediate mode. When used with the STOP, GOTO, and PRINT statements and the CONT and CCONT commands, immediate mode lets you debug programs without having to run them over and over. Use these BASIC–PLUS debugging features to test and correct source programs before you save them in translated form.

In addition to showing how to debug a source program in immediate mode, this chapter shows some other uses of immediate mode, describes its limitations, and also describes the "Ready state." A job is in the Ready state when the system is waiting for terminal input. Understanding the Ready state can help you debug programs.

RSTS/E provides another tool for debugging BASIC–PLUS programs, the BPCREF utility program. BPCREF, the last topic presented in this chapter, produces a cross-reference listing of a translated BASIC–PLUS program. BPCREF is useful for debugging large and complex programs.

Unlike the previous five chapters, which are mainly for new BASIC–PLUS users, this chapter is written for both new users and users who are familiar with the BASIC–PLUS language and have written some programs. New users may want to read just the section on immediate mode and the example in Section 6.3.5 to become familiar with the mechanics of debugging. The rest of the chapter, which describes the Ready state, the debugging process, and the BPCREF program, is primarily for the more experienced user.

6.1 Immediate Mode

Immediate mode lets you execute BASIC–PLUS statements without writing a complete program. When you type a statement without a line number, BASIC–PLUS treats it like a command, translating and executing it immediately. On the other hand, when you type a statement with a line number, BASIC–PLUS translates it and stores it for later execution. Immediate mode is useful for two different purposes: debugging programs and performing simple calculations at your terminal.

6.1.1 Immediate Mode Examples

The following example shows how BASIC–PLUS treats the same statement typed with and without a line number. Type:

```
10 PRINT 54+46RET
```

This statement produces no output on your terminal. The cursor simply moves to the next terminal line and waits for more input. Now type:

```
PRINT 54+46RET
```

BASIC–PLUS prints the sum of the two numbers on your terminal and then displays the Ready prompt:

```
 100

Ready
```

You can execute only one statement at a time in immediate mode. Statements you type in immediate mode, however, can refer to variables defined in the current program or in other immediate mode statements. For example:

```
A=3

Ready

B=4

Ready

PRINT A/B
 .75

Ready

PRINT SQR (A^2 + B^2)
 5

Ready
```

You can print a table of square roots with the following immediate mode statement:

```
PRINT I,SQR(I) FOR I = 1 TO 10
 1             1
 2             1.41421
 3             1.73205
 4             2
 5             2.23607
 6             2.44949
 7             2.64575
 8             2.82843
 9             3
 10            3.16228
```

6.1.2 Variable Assignments

A value you assign to a variable with an immediate mode statement lasts until either:

- You change its value with another immediate mode statement
- A statement in your current program changes its value

BASIC–PLUS sets all variables to 0 or the null string when you use the NEW, OLD, or RUN commands, enter a CCL or DCL command, switch to another keyboard monitor, or log out.

6.1.3 Limitations of Immediate Mode

Some BASIC–PLUS statements cannot be executed in immediate mode. These statements cannot stand alone; instead, they must interact with other statements in a program to produce results. These statements are:

DEF
DEF*
FNEND
DIM
DATA
FOR
WHILE
UNTIL
NEXT

When you type these statements in immediate mode, BASIC–PLUS displays the message:

```
?Illegal in immediate mode
```

Note that while FOR, WHILE, and UNTIL cannot be used as statements in immediate mode, they can be used as *statement modifiers*. The example in Section 6.1.1 that prints a square root table uses FOR as a statement modifier; see also Section 13.6.

In programs, you can place more than one statement on a single line by using the backslash (\) between statements. However, you cannot enter multiple statements on a single line in immediate mode. You get an error message and the statements are not executed. For example:

```
A=1,\ PRINT A
?Illegal in immediate mode

Ready
```

6.2 Background for Debugging – The Ready State

When the system is waiting for input from a job's keyboard, the job is said to be in the *Ready state*. When you work in BASIC–PLUS, you can usually, but not always, recognize the Ready state by the BASIC–PLUS "Ready" prompt on your terminal.

Most of the time, you only need to know that BASIC–PLUS expects input from you. But a more complete understanding of the Ready state can be helpful, especially during program debugging. Many commands and statements that you use during program debugging place your job in the Ready state. So do certain error conditions that occur during program execution. Thus, this section is presented as an introduction to debugging, and the specifics of debugging are described in the sections that follow.

NOTE

This description of the Ready state discusses BASIC–PLUS statements and concepts that are covered later in this manual. Cross-references to chapters and sections are provided. If you do not yet know the BASIC–PLUS language, you may want to postpone reading this section until you learn more about the language.

In BASIC–PLUS, a job enters the Ready state when:

- A program runs to completion and exits.
- BASIC–PLUS finishes executing a command or immediate mode statement.
- BASIC–PLUS finishes translating a line of code you just entered into the current program. (BASIC–PLUS does not display the Ready prompt after it translates a line of code.)

A job also enters the Ready state when a program's execution is halted before it is complete, either by the user or by the system. Halting a program at various points during execution to examine variables or change a program is an integral part of program debugging in BASIC–PLUS.

The Ready state is also called the "CTRL/C state" or the "keyboard monitor wait state." You may see these terms in other manuals. The SYSTAT and CTRL/T displays use "^C" to denote the Ready state.

The rest of this section describes:

- How a job executing a program enters the Ready state
- What information about the program that was executing (called "program context" or "context information") is saved
- What to do next (for example, how to continue executing a program that was halted)

6.2.1 Entering the Ready State

A job executing a program enters the Ready state when the program runs to completion. In addition, certain other events cause the system to halt the program before execution is complete and place the job in the Ready state. This action occurs when the system:

- Executes a STOP statement in the program
- Receives a CTRL/C from the keyboard
- Encounters an "untrapped" error, that is, one the program does not process using an ON ERROR GOTO statement

The STOP statement and CTRL/C, which are described in Section 6.3, give you control over program execution. STOP, an important BASIC–PLUS debugging feature, lets you halt a program at specific points during execution to examine or change data values or modify the program.

Pressing CTRL/C while a program is running also halts execution. For example, you need to press CTRL/C to halt a program that contains a logic error called an *infinite loop*. An infinite loop is a section of code that causes the program to keep executing the same sequence of instructions indefinitely. Use CTRL/C with caution, though, because you cannot control where the program will halt.

The system halts a program during execution when it encounters an error condition for which the program has no error-handling code. For example, when you run a program that tries to open a nonexistent file with an OPEN FOR INPUT statement, the "?Can't find file or account" error results. If the program contains code to process the error, the system provides the program with information about the error and gives control to the program. (See Section 13.7, Error Handling, for information on how to process errors in a program.) If the program does not contain code to process the error, the system notifies you instead. It displays a message on your terminal and places your job in the Ready state. In this example you see:

```
?Can't find file or account at line n

Ready
```

6.2.2 Program Status

BASIC–PLUS saves information about the program that was executing and sometimes performs "housekeeping" actions when it places a job in the Ready state. The information saved and the actions performed depend on:

- Whether or not the program is privileged
- What caused the job to enter the Ready state

If the program is privileged (that is, the program is stored on disk with a protection code that includes the value 128), BASIC–PLUS deletes the program from memory before it places the job in the Ready state. See the *RSTS/E Programming Manual* for more information about privileged programs. The rest of this discussion applies to nonprivileged programs.

BASIC–PLUS always saves the current values of program variables when a job running a nonprivileged program enters the Ready state. Program variables are saved when a program runs to completion; they are also saved after BASIC–PLUS executes an immediate mode statement. (During debugging, you can change the values of program variables by typing immediate mode statements.)

Besides saving variables, BASIC–PLUS also closes all I/O (input/output) channels after program execution is complete. An I/O channel is a logical connection between your program and a file or device. Closing I/O channels closes files and frees devices for other use. See Chapter 14, Overview of Data Handling, for more information about I/O channels.

BASIC–PLUS closes I/O channels when it executes either an explicit or an implicit END statement. An explicit END statement appears in the program. BASIC–PLUS executes an implicit END statement when the END statement does not appear in the program. (You can also type an END statement in immediate mode.)

When a STOP statement, a CTRL/C, or an untrapped error halts a program before execution is complete, BASIC–PLUS saves current values of program variables, but does not close I/O channels. In addition:

1. When CTRL/C halts a program, the BASIC–PLUS variable LINE contains the line number where the halt occurred.
2. When an untrapped error halts a program:
   - The BASIC–PLUS variable ERR contains the error number that halted the program.
   - The BASIC–PLUS variable ERL contains the line number where the error occurred.

   See Section 13.7 for more information about error handling and Appendix C for a list of error messages and their ERR values.

6.2.3 Possible Actions

You can perform many different actions when your job is in the Ready state. These actions have different effects on your program context.

If you are in the Ready state because a program ran to completion, or you do not want to continue executing the program that halted, you can either:

- Enter an immediate mode statement
- Enter a BASIC–PLUS command
- Modify the program by changing lines or entering new lines
- Run another program
- Enter a system command (CCL or DCL) or run a system program

Some BASIC–PLUS commands (such as NEW, OLD, and RUN) and all CCL and DCL commands delete your current program from memory, destroying all program context. Other BASIC–PLUS commands, such as LIST and SAVE, do not affect your current program. Immediate mode statements do not delete your current program, but do change the values of any variables that you specify.

If you are in the Ready state because of a program halt, you can continue executing the halted program. How to continue depends on what caused the program to halt.

If a STOP statement or a CTRL/C halted the program, you can either:

- Enter the CONT command or the privileged CCONT command to continue executing the program at the point where it halted. (See Sections 6.3.2 and 6.3.3.)
- Enter an immediate mode GOTO statement to continue execution at a specified line number. (See Section 6.3.4.)
- Enter an immediate mode END statement to end the program and close open I/O channels.

If an untrapped error halted the program, you can either:

- Enter an immediate mode GOTO statement to continue execution at a specified line number.
- Enter an immediate mode END statement to end the program and close open I/O channels.

You cannot use CONT or CCONT to continue executing a program that halted because of an untrapped error. BASIC–PLUS displays the message "?Can't CONTinue."

6.2.4 Summary

Table 6–1 summarizes the information presented in this section.

Table 6–1: Program Status in the Ready State

Cause of Halt	Program Status	Continuing or Ending Execution
END	Variables saved. I/O channels closed.	
STOP	Variables saved. I/O channels open.	CONT or CCONT Immediate Mode GOTO Immediate Mode END
CTRL/C	Variables saved. I/O channels open. LINE contains line number where error occurred.	CONT or CCONT Immediate Mode GOTO Immediate Mode END
Untrapped Error	Variables saved. I/O channels open. ERR contains error number. ERL contains line number where error occurred.	Immediate Mode GOTO Immediate Mode END
Any cause; privileged program	Program deleted from memory. I/O channels closed.	

The rest of this chapter describes in detail the BASIC–PLUS and RSTS/E features for debugging programs.

6.3 Debugging in Immediate Mode

Use of STOP statements in a program, combined with the use of CONT and CCONT commands and immediate mode GOTO statements, lets you:

- Halt a program's execution
- Examine variables
- Make any necessary changes to the program
- Continue program execution at the next line or another location

This section describes the use of these BASIC–PLUS debugging features.

6.3.1 The STOP Statement

The STOP statement halts program execution and prints a message containing the line number where the halt occurred. For example, if you place a STOP statement on line 50 in a program, the following message is displayed when the program is executed:

```
Stop at line 50

Ready
```

The program is no longer running. You can now examine data values with immediate mode PRINT statements or change data values with immediate mode LET statements. If you are working with a source program (as you usually are while debugging) you can also add, delete, or change lines with BASIC–PLUS editing features.

6.3.2 The CONT Command

After you halt execution and examine or change your program, you can use the the CONT command to continue program execution. Enter the command as:

```
CONT
```

The program resumes execution at the line after the STOP statement.

Usage Note

Certain error conditions can prevent the CONT command from continuing program execution. If execution cannot be continued, the system displays:

```
?Can't CONTinue

Ready
```

Depending on the cause of the error, run the program again, type an immediate mode GOTO statement to continue execution, or type an immediate mode END statement to end the program.

6.3.3 The CCONT Command

The CCONT command, available to privileged users only, performs the same action as CONT but also detaches the job from the terminal. Use it to resume execution of a job that needs to run for a long time without further terminal interaction. To enter the CCONT command, type:

```
CCONT
```

If you are not a privileged user, the system displays the message "?Protection violation." Use the CONT command to continue executing the program.

6.3.4 The GOTO Statement

Instead of continuing execution at the next line in the program after a halt, you can use the GOTO statement in immediate mode to resume execution at any line in the program. For example, enter the following statement to resume execution at line 100 of the program:

```
GOTO 100
```

Be sure to specify the correct line number. Any GOTO statement that causes transfer of control into or out of a FOR loop, function, or subroutine can cause unexpected results.

Note that when a program halts because of an untrapped error, you can resume execution only with the GOTO statement, not with CONT or CCONT. See Section 13.7 for information on error handling.

6.3.5 Debugging Example Using STOP and CONT

The following example uses an octal-to-decimal conversion program that contains a bug. Assume that you have written the program. When you run it, you find that it returns incorrect results; for example, it converts the octal number 23 to the decimal number 40 instead of 19. You decide to debug the program using STOP statements, the CONT command, and immediate mode. A program listing is on the next page. The step-by-step procedure to debug the program appears after the program listing.

This example uses a fairly advanced BASIC–PLUS program. The program contains several functions, a FOR loop, and an error handling routine. The lines of text that start with an exclamation point (!) are called comments. Comments describe the program but do not affect its execution. Even if you do not fully understand how the program works, you can still step through the debugging procedure to become familiar with the debugging process.

```
10      !This is an octal to decimal conversion program, &
        !It is written in extend mode.
20      EXTEND
100     ON ERROR GOTO 900 &
        \ PRINT                                     ! print out the &
        \ PRINT "OCTAL TO DECIMAL CONVERTER"        ! instructions &
        \ PRINT "CONVERTS NUMBERS BETWEEN 0"        ! header &
        \ PRINT "AND 177777 (OCTAL) TO" &
        \ PRINT "THEIR DECIMAL EQUIVALENTS" &

200     INPUT "OCTAL NUMBER";S$                     ! input char string &
        \ L% = LEN(S$)                              ! get its length &
        \ GOTO 600 IF L% = 0%                       ! close out if &
                                                    ! length is 0.
210     !Starting with low order digit, &
        !for each digit, take value of digit (V%), &
        !multiply it by 8 to proper order (O%), &
        !and add result to accumulator (D%) &

300     O% = -1%                                    ! initialize order &
        \ D% = 0%                                   ! and accumulator &
        \ FOR Z% = L% TO 1% STEP -1%                ! for each digit from &
                                                    !  the low order to &
                                                    !  the high order &
        \       O% = O% + 1%                        ! increment order &
        \       V% = ASCII(RIGHT(S$,Z%))            ! pick up next digit &
        \       IF V% < 48% OR V% > 55% THEN        ! if out of range: &
                        PRINT "INVALID INPUT"       !  print message &
        \               GOTO 200                    !  ask for more input &
```

(continued on next page)

```
400             V% = V% AND 7%                      ! change ascii &
        \       D% = D% + (V% * INT(8% ^ O%))       ! to a numeric &
                                                    ! and the value will &
                                                    ! accumulate in D%. &
                                                                        &
        \ NEXT Z%                                   ! go back and do next &
                                                    ! digit if there &
                                                    ! is one. &

500     PRINT "DECIMAL VALUE IS ";NUM1$(D%)         ! print the result &
        \ GOTO 200                                  ! and try another. &

600     GOTO 32767

700     ! This error routine handles numbers too large to convert.
900     IF ERL = 400 THEN &
                PRINT "NUMBER ";S$;" TOO BIG FOR CONVERSION" &
        \       RESUME 200
910     PRINT ERR,ERL &
        \       RESUME 32767
32767   END
```

1. You suspect that the bug is in line 300 or 400 and therefore insert a STOP statement before each of these lines, at lines 250 and 350:

```
250 STOP
350 STOP
```

   If you now list the program, you see that it contains the STOP statements you inserted.

2. Now run the program and input the octal number 23. The program stops at line 250:

```
RUNNH

OCTAL TO DECIMAL CONVERTER
CONVERTS NUMBERS BETWEEN 0
AND 177777 (OCTAL) TO
THEIR DECIMAL EQUIVALENTS
OCTAL NUMBER? 23
Stop at line 250

Ready
```

3. Check the value of the variable L%, which contains the number of digits in the octal number you input. Issue a PRINT statement in immediate mode:

```
PRINT L%
 2

Ready
```

4. The value of L% is correct; 23 has 2 digits. Type the CONT command to continue program execution. The program stops at the next STOP statement.

```
CONT
Stop at line 350

Ready
```

5. Now check the values of the variables D%, O%, V%, and Z%. Issue another PRINT statement in immediate mode:

```
PRINT D%,O%,V%,Z%
 0             1             51            2
```

You see that the value of O%, printed as 1, should be 0 at this point in the program. (The program is using the digit in position 0 and O% was initialized to –1.) You list the program and find the error in the sixth text line at line 300, where O% (the letter variable that stands for "order") was mistyped as 0% (zero).

6. If you wish, you can now retype line 300 or use a text editor to correct the typing error and then run the program again:

```
300     O% = -1%                                    ! initialize order &
        \ D% = 0%                                   ! and accumulator &
        \ FOR Z% = L% TO 1% STEP -1%                ! for each digit from &
                                                    !   the low order to &
                                                    !   the high order &
        \       O% = O% + 1%                        ! increment order &
        \       V% = ASCII(RIGHT(S$,Z%))            ! pick up next digit &
        \       IF V% < 48% OR V% > 55% THEN        ! if out of range: &
                        PRINT "INVALID INPUT"       !   print message &
        \               GOTO 200                    !   ask for more input &

RUNNH

OCTAL TO DECIMAL CONVERTER
CONVERTS NUMBERS BETWEEN 0
AND 177777 (OCTAL) TO
THEIR DECIMAL EQUIVALENTS
OCTAL NUMBER? 23
DECIMAL VALUE IS 19
```

After making one or more corrections, use the REPLACE command to save the latest version of your program in a disk file. See Section 5.3.

6.4 Halting and Checking Execution – CTRL/C and PRINT LINE

Pressing CTRL/C while a program is running also halts program execution. However, CTRL/C gives you less control over where the program halts than the STOP statement because you cannot specify where the halt is to occur.

When a program is halted by CTRL/C, the special variable LINE contains the line number of the statement being executed at the time of the halt. After using CTRL/C, you can display the contents of LINE by typing a PRINT LINE statement in immediate mode:

```
^C

Ready

PRINT LINE
 300

Ready
```

Use the CONT command, the CCONT command, or an immediate mode GOTO statement to continue program execution.

Notes

1. If a multi-statement line is being executed when CTRL/C interrupts a program, you cannot determine where in the line the program stopped.
2. Special programming techniques are available to prevent program interruption by CTRL/C. See the *RSTS/E Programming Manual.*

6.5 Controlling Terminal Output

Sometimes you may want to stop and restart output to your terminal while a program is running without stopping program execution. Several control keys allow you to do this.

6.5.1 Stopping Output with CTRL/O

CTRL/O stops output to the terminal without stopping program execution. The program continues to produce output, but the system discards it. The system often, but not always, displays the Ready prompt when the program is finished.

If you want to resume terminal output before the program is finished running, press CTRL/O again.

A program cannot determine whether a CTRL/O has been entered at the terminal. However, there is a system function call that cancels the effect of a CTRL/O that has been entered. Refer to the *RSTS/E Programming Manual* for more information.

6.5.2 Suspending and Resuming Output with CTRL/S and CTRL/Q

If your terminal has the STALL characteristic set (most video terminals do), you can use CTRL/S and CTRL/Q to suspend and resume terminal output while a program is running. Press CTRL/S to suspend output; press CTRL/Q to resume it.

The system handles CTRL/S and CTRL/Q differently than CTRL/O. The system saves program output after you press CTRL/S instead of discarding it. Thus, all output is displayed after you press CTRL/Q. If the program finishes executing after you press CTRL/S, the system does not display the Ready prompt until you press CTRL/Q.

(See the *RSTS/E System User's Guide* for information about the STALL characteristic.)

6.5.3 Suspending and Resuming Output with the NO SCROLL Key

The NO SCROLL key on the VT100 terminal performs the same function as CTRL/S and CTRL/Q. The terminal STALL characteristic must be set. Press NO SCROLL to suspend terminal output while a program is running; press NO SCROLL again to resume terminal output.

(See the *RSTS/E System User's Guide* for information about the STALL characteristic.)

6.6 Getting a Cross-Reference Listing – The BPCREF Program

The BPCREF (BASIC–PLUS cross reference) program is a useful debugging tool for large and complex programs. BPCREF produces a cross-reference listing of a translated BASIC–PLUS (.BAC) program. The default listing includes variables, line number references, and some statistical data. You can ask BPCREF to include the source file in the listing, print local or global references, and queue the output to the line printer spooler.

The references in the BPCREF listing can help you find errors in your program and assist you in modifying your code. For example, local and global references can be useful when you segment a BASIC–PLUS program into chainable modules or convert a BASIC–PLUS program to BASIC–PLUS–2.

BPCREF works only on BASIC–PLUS translated images (that is, .BAC files). It does not work on BASIC–PLUS source files or BASIC–PLUS–2 object code or task images.

6.6.1 Running BPCREF

Check with your system manager to find out where BPCREF is installed. If it is installed in the system library account, type the following command to run it:

```
RUN $BPCREF
```

BPCREF prints a header line and prompts you for input:

```
BPCREF command?
```

You need to specify the files to be processed. Type the command in the format:

```
output.crf=input.bac [,input.bas] [/switches]
```

where:

output.crf	is the cross-reference listing file.
input.bac	is the translated file to process.
input.bas	is the source file to include in the listing.
/switches	are one or more switches that specify the action to be taken on the files.

You need not specify the entire command line as shown; there are defaults for both file names and types. Note that you can use the /SOURCE switch to include a listing of the .BAS file if it has the same file name and is located in the same account as the .BAC file. Otherwise, you must specify the source file in the BPCREF command. Table 6–2 shows several ways to enter the BPCREF command and describes the result.

Table 6–2: BPCREF Command Formats

Format	Result
file1	BPCREF examines file1.BAC and produces an output file with the same name and the file type .CRF.
file1=file2	BPCREF examines file2.BAC and produces an output file named file1.CRF.
file1,file2	BPCREF examines file1.BAC; includes the source listing of file2.BAS in the output file file1.CRF.
file1=file2,file3	BPCREF examines file2.BAC; includes in the output file file1.CRF the source listing of file2.BAC, which is in file3.BAS.

The BPCREF command switches, listed in Table 6–3, let you print the listing on the line printer and tailor its format and contents. You can abbreviate switches to one or more characters. Table 6–3 shows the required number of characters for each switch; optional characters are inside square brackets. If you use an undefined or incorrect switch in a command that does not include the /QUEUE switch, BPCREF prints a warning message and ignores the undefined switch. If you use an undefined or incorrect switch in a command that includes the /QUEUE switch, BPCREF passes the undefined switch to the spooling program and notifies you with a message.

When you use the /QUEUE switch, the output file is queued to the default line printer, LP0:. However, you can direct BPCREF output to a different printer by specifying it as the output file, for example, LP1:=file1/Q or LP1:=file1. The first command queues file1 to LP1:; the second command sends file1 directly to LP1:.

Table 6–3: BPCREF Command Switches

Switch	Format
/HEL[P]	Displays information about BPCREF command formats.
/NOD[ELETE]	Tells the spooling program not to delete the output files after printing.
/NOH[EADER]	Omits header lines in the output file. The resulting .CRF file is more suitable as input to the FILCOM program.
/P[AGE]:nnn	Specifies nnn as the number of lines per page to print in the output listing. Sixty lines per page is the default.
/Q[UEUE]	Automatically queues the output file to the line printer. The output file is deleted after printing unless you use the /NODELETE switch.
/S[OURCE]	Includes the source file in the output file. Use this switch when the .BAS file is in the same account and has the same file name as the .BAC file. Otherwise, specify the source file in the BPCREF command.
/W[IDTH]:nnn	Sets the width of the cross reference listing to nnn columns per line. The default width is 132 columns; the minimum width is 72 columns.
/GL[OW]:nnn /GH[IGH]:nnn	Used as a pair, these switches specify a global listing. BPCREF prints line numbers and variables referenced both inside and outside the program boundaries indicated by the line numbers nnn. BPCREF does not print line numbers or variables referenced only inside or outside the boundaries. The default value for /GL is 0 and the default value for /GH is 32767.
/LL[OW]:nnn /LH[IGH]:nnn	Used as a pair, these switches specify a local listing. BPCREF prints the line numbers and variables referenced inside the program boundaries indicated by the line numbers nnn. BPCREF does not print line numbers or variables referenced outside the boundaries or referenced both inside and outside the boundaries. The default value for /LL is 0 and the default value for /LH is 32767.
/NOC[REF]	Omits the full cross reference listing. Use it when you want only a local or global listing.

6.6.2 Output Listing Contents

BPCREF produces an output file that contains:

- An ordered and cross-referenced list of line numbers and variables in the translated image and the line numbers on which they are referenced
- A list of line numbers and variables whose use is questionable
- A statistical summary

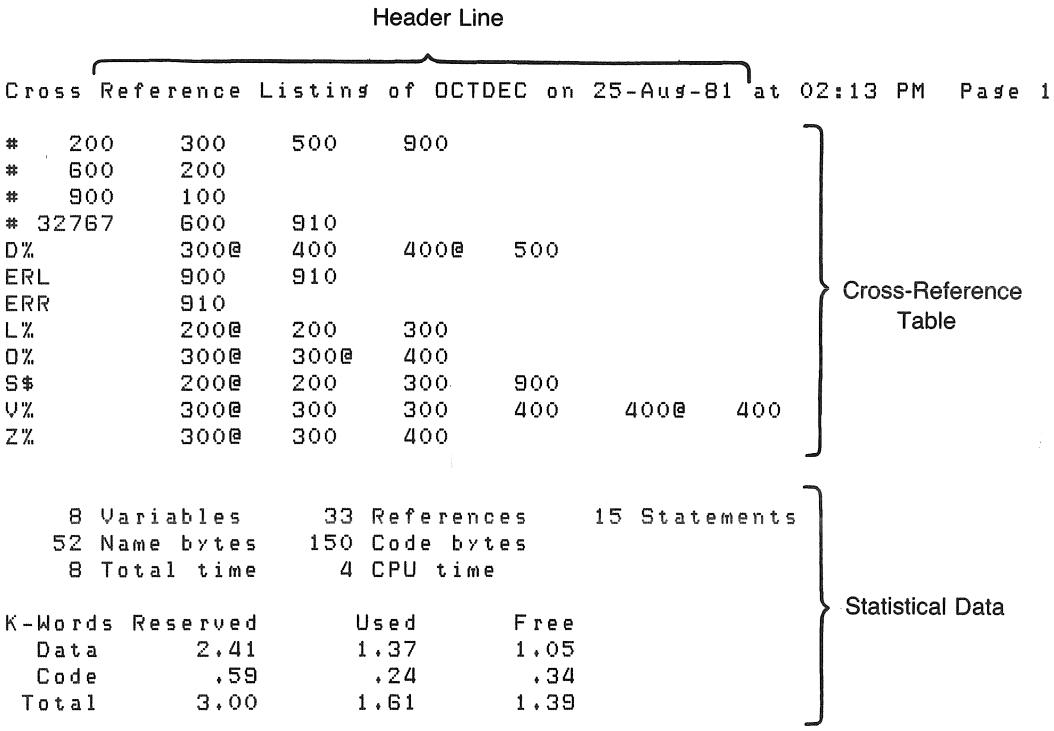
If you include the source file in your listing, it is printed at the beginning of the output file.

Figure 6–1 shows a cross-reference listing of the octal-to-decimal conversion program used in the debugging example. (The bug was not corrected.) The following command was used to produce this listing:

```
BPCREF command? OCTDEC/WIDTH:72
```

The /WIDTH switch set the listing's width to 72 columns. BPCREF placed the output in the file OCTDEC.CRF.

Figure 6–1: Sample Cross-Reference Listing



The following sections describe each part of the listing.

6.6.2.1 Header Line

Each page of the output file has a header line that contains the translated program name, the current date and time of day, and a page number. Use the /NOHEADER switch to omit this line.

6.6.2.2 Cross-Reference Table

A cross-reference table of line numbers and variables is printed under the header line. Lines that start with a number sign (#) show references to line numbers made in the program. If no lines in the program reference other lines, this information is omitted.

For example:

```
#   200     300     500     900
```

This line means that line number 200 is referenced in lines 300, 500, and 900 of the OCTDEC program. Each of these lines contains a GOTO 200 statement.

BPCREF prints cross-references to variables below the cross-references to line numbers. Each line consists of a variable name followed by one or more line numbers. The following codes (printed after line numbers) show the kind of references to variables:

@	Destructive reference	The reference to the variable on the line changed the value of the variable.
#	Definition	A DIM statement or function defines the variable.

For example:

```
D%          300@    400     400@    500
```

This line means that the variable D% is referenced once in lines 300 and 500 and twice in line 400. The value of D% changes in references marked with the @ symbol.

6.6.2.3 Statistical Data

The last part of BPCREF's output, the statistical data, includes:

- The number of variables in the code
- The number of references to line numbers and variables
- The number of statements in the code
- The number of bytes needed to store variable definitions (name bytes)
- The number of internal code elements (code bytes)
- The elapsed and processor time (in seconds) used to create the output

BPCREF also prints the size (in K words) of the program's data and code areas, and the amount of memory reserved, actually used, and free for program expansion.

6.6.2.4 Optional List of Suspect Line Numbers and Variables

BPCREF determines, by cross-checking references to variables, whether a variable may be incorrectly referenced (never read, never written, never defined) and prints a list of these variables. The optional list of suspect line numbers and variables is introduced by the line:

```
Please check that the following variables have been referenced properly:
```

This list points out possible typing errors in the source code. Note that BPCREF cannot logically analyze the program but can find simple typing and logic errors.

BPCREF finds no suspect line numbers or variables in the octal-to-decimal conversion program. However, the following program contains typing and logic errors that BPCREF can find. Line 200 references line 90, which does not exist, and two variable names, RADIUS and VOLUME, are misspelled. Because VOLUME is misspelled, its value is not calculated. A cross-reference listing that shows the suspect line numbers and variables follows the program text.

```
100 EXTEND
200 ON ERROR GOTO 90                        ! Error here
300 INPUT "Radius"; RADIUS
400 AREA   = PI * RADIUS ^ 2.0
500 VOLUEE = PI * RADIAS ^ 3.0              ! Errors here
600 PRINT "Area ="; AREA; ", Volume ="; VOLUME
700 GOTO 300
999 END
```

```
   Cross Reference Listing of DEMO on 25-Aug-81 at 02:30 PM      Page 1

#     90      200
#    300      700
AREA          400@    600
PI            400     500
RADIAS        500
RADIUS        300@    400
VOLUEE        500@
VOLUME        600

Please check that the following variables have been referenced properly:
#     90  RADIAS   VOLUEE   VOLUME
```

6.6.2.5 Global and Local Listings

Global and local listings let you divide a program into two regions and print listings of variable references that indicate whether the reference is inside one region (local) or common to both (global). Such listings are useful if you plan to segment the program into chainable modules or convert the program to BASIC–PLUS–2.

To request a global or local listing, use the global (/GL and /GH) or local (/LL and /LH) switches on the BPCREF command line. Global and local listings follow the full cross-reference listing. Each global or local listing you request appears on a separate page. Statistical information is printed after global and local listings. (If you want only a local or global reference listing, specify the /NOCREF switch to omit the full reference listing.)

You can combine the global and local switches to get a listing of both types of references. For example:

```
BPCREF command? OCTDEC/GL:300/GH:400/LL:300/LH:400/W:72
```

This command adds the following information to the sample listing of the octal-to-decimal conversion program:

```
  Cross Reference Listing of OCTDEC on 25-Aug-81 at 02:19 PM     Page 2

                  Global range is 300 to 400

#    200     300     500     900
D%           300@    400     400@    500
L%           200@    200     300
S$           200@    200     300     900
```

This global listing means that line number 200 and the variables D%, L%, and S$ are referenced both inside and outside the global range, line numbers 300 through 400 in the program. References marked with the @ symbol are destructive. All other variables in lines 300 through 400 are referenced only inside the global range.

```
  Cross Reference Listing of OCTDEC on 25-Aug-81 at 02:19 PM     Page 3

                  Local range is 300 to 400

O%        300@    300@    400
V%        300@    300     300     400     400@    400
Z%        300@    300     400
```

This local listing means that the variables O%, V%, and Z% are referenced only inside the local range 300 through 400. References marked with the @ symbol are destructive. No line numbers are locally referenced in this range.

When more than one global or local switch of the same boundary type appear on the command line, BPCREF uses the last specification. For example:

```
BPCREF command? OCTDEC/GL:30/GL:20
```

BPCREF uses line number 20 as the global reference lower boundary.

6.6.3 Error Messages

BPCREF displays error messages if you give incorrect input or if it finds fatal or unexpected conditions during processing. The messages are listed and described in Table 6–4. The first character of each message shows the severity of the error. The % character indicates a warning; BPCREF continues processing but may not produce the results you expect. The ? character indicates a fatal error; BPCREF cannot finish processing and returns you to system command level.

Table 6–4: BPCREF Error Messages

?BPCREF *version* fatal error *mmm* at line *nnn – text*

BPCREF or its auxiliary module BPCRF1 encountered the condition described by RSTS/E error number mmm while executing an operation at line number nnn. BPCREF cannot continue. (Error numbers are listed in Appendix C.)

?Can't open *filename – text*

BPCREF cannot access the named file because of the condition described by the RSTS/E error text.

?Command error – *text*

While processing the command you entered, BPCREF encountered the RSTS/E error described in the text. BPCREF prints a help message and redisplays the BPCREF Command? prompt.

%Correct prime – *nnn*

BPCREF has been incorrectly modified to change the variable table size. See your system manager.

?Could not chain to *filename – text*

BPCREF must chain to the auxiliary module BPCRF1 described by filename. The RSTS/E error is printed in text. See your system manager.

%Illegal width, minimum = 72

You specified a value in the /WIDTH switch that is less than the allowed minimum. BPCREF sets the width to the minimum (72 columns) and continues processing.

?*filename* is not compiled BASIC–PLUS program

The file indicated by filename is not a valid BASIC–PLUS translated image. BPCREF cannot process a source file, task image, or any other type of file.

?Maximum memory exceeded

Because of the high number (or great length) of variable names, the BPCREF variable table overflowed. Subdivide the program and process each module separately.

?*filename* must be on disk

The file named in filename is not on a disk device. BPCREF processes only disk files.

?Please run BPCREF

You tried to run the BPCRF1 module, which is part of the BPCREF program. BPCRF1 prints this message and returns you to system command level.

%Reference table full

An internal table is full. BPCREF continues processing but loses some data. Subdivide the program and process each module separately.

%Unknown compiled code *nn* at line *mm*

BPCREF encountered a BASIC–PLUS operation code (nn) that it did not recognize. BPCREF continues processing but it may lose some data or output incorrect data (for example, the program may miss a variable reference at this point).

%Unknown switch(es), *text*

You specified an undefined switch in a BPCREF command without the /QUEUE switch. BPCREF ignores the switch and continues processing.

(continued on next page)

Table 6–4: BPCREF Error Messages (Cont.)

%Unknown switch(es), *text* will be passed to queue manager You specified an undefined switch in a BPCREF command containing the /QUEUE switch. BPCREF passes the information in "text" to the spooling program.
%Variable table full BPCREF can process a maximum of 421 variables (plus line numbers used in GOTO and GOSUB statements). Decrease the number of variables in the program and try again, or subdivide the program and process each module separately. Note that the system manager can increase the maximum number of variables that BPCREF can process.

PART II
Elementary Language Features

Chapter 7
Building a Program

This chapter describes the structure and format of BASIC–PLUS programs. It describes line numbers, statements, remarks, and comments, and explains the difference between EXTEND and NOEXTEND format.

7.1 Sample BASIC–PLUS Program

The program in Figure 7–1 is written in the BASIC–PLUS language, using EXTEND format. It illustrates the syntax of the language, the format of statements, and the appearance of terminal output.

The program calculates totals from random dice rolls. When you run the program, you enter the number of dice per roll and the number of rolls.

7.2 Parts of a Program

A BASIC–PLUS program consists of lines of statements that contain instructions. Each line starts with a line number.

7.2.1 Line Numbers

Line numbers, which can range from 1 to 32767, specify the order in which statements are to be executed. You can enter lines in any order when you type the program into the system; BASIC–PLUS always keeps the program in order by line number.

Figure 7–1: Sample BASIC–PLUS Program

```
LISTNH
50 EXTEND
100 RANDOMIZE &
    !THIS IS A RANDOM DICE ROLL ROUTINE &
    !THE USER CAN SPECIFY HOW MANY DICE TO BE IN &
    !EACH ROLL AND HOW MANY ROLLS ARE TO BE MADE, &
    !WHETHER TO PRINT THE TOTAL OF EACH ROLL IS ALSO &
    !UNDER USER CONTROL
110 PRINT 'THIS PROGRAM GIVES RANDOM DICE ROLLS' &
    \ PRINT 'HOW MANY DIES IN EACH ROLL'; &
    \ INPUT N &
    \ PRINT 'HOW MANY ROLLS'; &
    \ INPUT D &
    \ PRINT 'IF YOU WANT THE TOTAL OF EACH ROLL, TYPE Y'; &
    \ INPUT R$ &
    \ PRINT
120 FOR J = 1 TO D &
    \ PRINT 'THE';N; 'DIES OF ROLL';J;'ARE:';
130 FOR I = 1 TO N &
    \ LET A%=(INT(6*RND) + 1) &
    \ LET B%=A% + B% &
    \ PRINT A%; &
    \ NEXT I
140 IF R$ = 'Y' THEN PRINT '---TOTAL OF ROLL =';B%
150 PRINT &
    \ LET B% = 0% &
    \ NEXT J
32767 END

Ready

RUNNH
THIS PROGRAM GIVES RANDOM DICE ROLLS
HOW MANY DIES IN EACH ROLL? 5
HOW MANY ROLLS? 3
IF YOU WANT THE TOTAL OF EACH ROLL, TYPE Y? Y

THE 5 DIES OF ROLL 1 ARE: 4  2  4  4  3 ---TOTAL OF ROLL = 17

THE 5 DIES OF ROLL 2 ARE: 6  5  1  4  5 ---TOTAL OF ROLL = 21

THE 5 DIES OF ROLL 3 ARE: 5  5  4  6  5 ---TOTAL OF ROLL = 25

Ready

RUNNH
THIS PROGRAM GIVES RANDOM DICE ROLLS
HOW MANY DIES IN EACH ROLL? 2
HOW MANY ROLLS? 5
IF YOU WANT THE TOTAL OF EACH ROLL, TYPE Y? N

THE 2 DIES OF ROLL 1 ARE: 2  3
THE 2 DIES OF ROLL 2 ARE: 4  4
THE 2 DIES OF ROLL 3 ARE: 4  3
THE 2 DIES OF ROLL 4 ARE: 3  6
THE 2 DIES OF ROLL 5 ARE: 3  5

Ready
```

Line numbers also:

- Let you change the order in which statements are executed. You can include statements in a program to make execution branch or loop around parts of the program. (See the sections on the GOTO, GOSUB, and IF–THEN statements in Chapter 9 for more information.)
- Help you debug and edit programs. BASIC–PLUS debugging features let you stop execution at any line in a program, examine variables and modify code, and then resume execution at any line. BASIC–PLUS editing features let you change one or more program lines without affecting the rest of the program. (See Chapters 5 and 6 in Part I for more information.)

When you first write a program, number lines in increments of 10 or more. This practice lets you insert new lines when debugging or enhancing the program. You may also want to start numbering at line 100 so you can enter a large amount of code at the beginning of your program at a later time.

7.2.2 Statements

You give instructions to BASIC–PLUS by writing statements. One or more BASIC–PLUS statements follows each line number in a program. The first word of each statement, called a *keyword*, identifies it, telling BASIC–PLUS which operation to perform and how to treat the rest of the statement.

The rest of a BASIC–PLUS statement consists of *operators* and *operands*. An operator is a symbol that specifies the action to perform (such as + for addition); an operand is the data on which to perform the action. Operands can be constants, which have fixed values, or variables, whose values can change during program execution. When you write statements, you combine operators and operands into *expressions*, which are groups of symbols that BASIC–PLUS evaluates. Chapter 8 describes keywords, constants, variables, operators, and expressions in more detail.

The sample program contains several different types of BASIC–PLUS statements, which in turn contain expressions composed of constants, variables, and operators. Consider these statements from line 130 of the sample program:

```
LET B%=A% + B%
PRINT A%;
```

The first statement, called an assignment statement, assigns a value to the variable B%. The part of the statement to the right of the equals sign is an arithmetic expression. It is composed of the variables A% and B% separated by a plus sign, which specifies addition. During program execution, BASIC–PLUS evaluates this expression by adding the current values of A% and B% and then assigns their sum to B%.

The next statement, a PRINT statement, prints the current value of the variable A% on the terminal. A% is the operand in this PRINT statement.

Other PRINT statements in the sample program have text as an operand. For example:

```
PRINT 'THIS PROGRAM GIVES RANDOM DICE ROLLS'
```

This statement prints on the terminal:

```
THIS PROGRAM GIVES RANDOM DICE ROLLS
```

You must enclose text in quotation marks when you write this kind of PRINT statement.

7.3 Line and Statement Formats

There are two types of lines in a BASIC–PLUS program: program lines and terminal lines, also called text lines. Program lines always have a line number. You can write a complete program line on one terminal line or you can continue a program line over several terminal lines. For example:

```
150     PRINT &(RET)
        \ B% = 0 &(RET)
        \ NEXT J(RET)
32767   END(RET)
```

Line 150 is a program line that is continued over three terminal lines. The first two terminal lines in line 150 end with an ampersand (&) followed by the RETURN key. This sequence indicates that the program line is continued on the next terminal line. The last terminal line in line 150 ends with just the RETURN key; this indicates the end of the program line. Line 32767 is a complete program line on one terminal line.

The simplest way to format a program is to place a single statement on each program line. However, BASIC–PLUS lets you place multiple statements on a single program line, and it also lets you continue a single statement on successive terminal lines.

7.3.1 Multi-Statement Lines

To write multiple statements on a single program line, end each statement (except the last) with a backslash (\).

The following examples show two ways to write a program line that contains three statements. In both examples, all three statements (LET, PRINT, and IF–THEN) are part of line 20 of the program. The first example uses a single terminal line; the second uses three terminal lines:

```
20 LET X=X+1.\ PRINT X,Y,Z\ IF Y=2. THEN GOTO 10(RET)
```

```
20      LET X=X+1. &(RET)
        \ PRINT X,Y,Z &(RET)
        \ IF Y=2. THEN GOTO 10(RET)
```

Writing each statement on a separate terminal line makes your program easier to read. In addition, if you use EXTEND mode, you can place a comment after each statement (see Section 7.4).

7.3.2 Multi-Line Statements

You continue a single BASIC–PLUS statement on successive terminal lines by ending each terminal line with the LINE FEED key (in both EXTEND and NOEXTEND modes) or an ampersand (&) character followed by the RETURN key (in EXTEND mode only). The ampersand/RETURN key combination is preferred. For example:

```
10 LET W7=(W-X4*3)*(Z-A/LF
(A-B)-17)RET

10 LET W7=(W-X4*3)*(Z-A/ &RET
(A-B)-17)RET
```

These statements are the same as:

```
10 LET W7=(W-X4*3)*(Z-A/(A-B)-17)RET
```

You cannot separate parts of a language keyword, a constant, a variable name, or a line number when you continue a statement. The following examples show correct and incorrect statement continuation. In the examples, A1 is a variable name and 100 is a line number. The first incorrect example inserts a break in the variable name; the second incorrect example inserts a break in the line number.

Correct

```
10 IF A1=0. &RET
THEN GOTO 100
```

Incorrect

```
10 IF A &RET
1=0. THEN GOTO 100RET
?Illegal conditional clause

10 IF A1=0. THEN GOTO 1 &RET
00
?Modifier error at line 10
```

Any illegal statement produces an error message.

BASIC–PLUS recognizes several phrases as single keywords that cannot be broken by a line continuation. These phrases are:

GO TO	AS FILE	MAT READ
GO SUB	FOR INPUT AS FILE	MAT PRINT
ON ERROR GO TO	FOR OUTPUT AS FILE	MAT INPUT
INPUT LINE	FILE SIZE	MAT LET
NO EXTEND		

Note that besides indicating continuation, the ampersand/RETURN key also signals the end of a comment. The LINE FEED key indicates continuation only; it does not signal the end of a comment. See Section 7.4 for more information.

7.3.3 Spaces and Tabs

Spaces and tabs make programs easier to read. Use spaces between line numbers and statements. Also use spaces between parts of a statement. For example:

```
10 LET X = Y*2 + 1

10 LET X=Y*2+1
```

These two LET statements are both valid, but the first is easier to read than the second.

You use spaces differently in EXTEND and NOEXTEND mode formats. See Section 7.5.2 for details.

The following example shows use of tabs:

```
2000     FOR K=1 TO 3
2010             FOR I=1 TO 10
2020                     FOR J=1 TO 10
2030                     A(I,J) = K/(I+J-1)+A(I,J)
2040                     NEXT J
2050             NEXT I
2060     NEXT K
32767    END
```

This example shows how tabs can make a program with nested FOR loops easier to read. FOR loops are used in programs to perform operations repetitively; each FOR loop starts with a FOR statement and ends with a NEXT statement. Nesting means placing one FOR loop inside another FOR loop. (FOR loops and nesting are explained in Chapter 9.)

7.4 Remarks and Comments

It is good programming practice to document a program as you develop it by including remarks and comments in the program. For example, the following kinds of information are useful to someone reading or modifying a program:

- The name and purpose of the program
- How to use the program
- How certain parts of the program work
- Expected results at various points in the program

You can insert this kind of information into a program with the REM statement and the exclamation point (!). Messages in REM statements are called remarks; those after the exclamation point are called comments. For example, line 100 of the program shown in Figure 7–1 contains several comments that describe the program:

```
100 RANDOMIZE &
    ! THIS IS A RANDOM DICE ROLL ROUTINE &
    ! THE USER CAN SPECIFY HOW MANY DICE TO BE IN &
    ! EACH ROLL AND HOW MANY ROLLS ARE TO BE MADE, &
    ! WHETHER TO PRINT THE TOTAL OF EACH ROLL IS ALSO &
    ! UNDER USER CONTROL
```

Remarks and comments are printed when you list your program but they do not affect program execution. They can, however, affect program size.

REM statements can contain any printing characters on the keyboard. In EXTEND mode, the word REM must be followed by a space or tab. BASIC–PLUS ignores, but includes in the line, anything that follows the letters REM. You can use the line number of a REM statement in a GOTO or GOSUB statement as the destination of a branch in program execution (see Sections 9.3 and 9.9). Typical REM statements are:

```
10 REM - THIS PROGRAM COMPUTES THE
11 REM - ROOTS OF A QUADRATIC EQUATION
```

Comments can be placed on the same line as statements. The exclamation point (!) ends the executable part of the line and begins the comment part of the line. When a line begins with an exclamation point, BASIC–PLUS treats the entire line as a comment. In the next example, BASIC–PLUS executes the LET statement on line 125 and the PRINT statement on line 130 but does not execute the PRINT statement on line 140:

```
125 LET A = 2+4*SQR(C)  !SET A EQUAL TO INITIAL VALUE
130 PRINT A/2 + 1  !PRINT SECOND CALCULATED VALUE
140 !COMMENT \ PRINT "THIS LINE IS A COMMENT"
```

In every statement except the DATA statement, BASIC–PLUS ignores everything on the line after the exclamation point. An exclamation point must not appear on the same program line as a DATA statement unless it is one of the items in the DATA statement list. The exclamation point becomes part of the data. (See Chapter 9 for a description of the DATA statement.)

The following examples show four ways of putting the same information on two lines. Lines 10 and 11 are REM statements. Line 20 is one REM statement broken into two lines with the LINE FEED key. Line 30 is one comment (begun with a "!") broken into two lines with the LINE FEED key. Line 40 is one comment broken into two lines with ampersand/RETURN, which is legal only in EXTEND mode.

```
10 REM THIS PROGRAM COMPUTES THE
11 REM ROOTS OF A QUADRATIC EQUATION

20 REM THIS PROGRAM COMPUTES THE (LF)
   ROOTS OF A QUADRATIC EQUATION

30 ! THIS PROGRAM COMPUTES THE (LF)
   ROOTS OF A QUADRATIC EQUATION

40 !THIS PROGRAM COMPUTES THE &(RET)
   !ROOTS OF A QUADRATIC EQUATION
```

Note that besides indicating continuation, the ampersand/RETURN key also signals the end of a comment. This EXTEND mode feature lets you alternate comments and code over several text lines. For example:

```
10 EXTEND
20 PRINT "This example"                    !Comment &(RET)
   \ PRINT "illustrates"                   !Comment &(RET)
   \ PRINT "how comments and statements"   !Comment &(RET)
   \ PRINT "can be used with &/RETURN in EXTEND mode."(RET)
30 END
```

When you run this program, it prints:

```
This example
illustrates
how comments and statements
can be used with &/RETURN in EXTEND mode.
```

BASIC–PLUS begins to recognize characters as statements instead of comments when it encounters the ampersand/carriage return. Thus, when you use this sequence to continue a comment on the next terminal line, you must start the continuation line with an exclamation point. Otherwise, the line will be interpreted as an executable statement. For example:

```
10 EXTEND
20 ! THIS PROGRAM COMPUTES THE &(RET)
   ! ROOTS OF A QUADRATIC EQUATION &(RET)
   PRINT "ENTER THREE COEFFICIENTS" (RET)
RUNNH
ENTER THREE COEFFICIENTS

Ready
```

Unlike the ampersand/RETURN key, the LINE FEED key does not signal the end of a comment. Thus, if you use the LINE FEED key for continuation instead of the ampersand/RETURN key (in either EXTEND or NOEXTEND mode), BASIC–PLUS treats all characters between the exclamation point and the end of the program line as a comment. For example:

```
10 PRINT "This is a statement,"  !Comment(LF)
   \ PRINT "but the rest of this program line is a comment."
20 END
```

When you run this program, only the first PRINT statement is executed. Thus, the program prints:

```
This is a statement
```

7.5 EXTEND and NOEXTEND Modes

BASIC–PLUS has two modes of operation: EXTEND and NOEXTEND. The mode you work in determines which BASIC–PLUS features you can use and how you must format your programs.

The examples in this manual are written to execute in EXTEND mode, the recommended mode of operation. This section explains how to change modes and describes the differences between EXTEND and NOEXTEND program formats.

7.5.1 Changing Modes

The system manager determines which mode is the default on each system. However, you can change modes at any time by using EXTEND or NOEXTEND as a command or a program statement. Enter EXTEND or NOEXTEND as a command to set the mode for the current terminal session; use EXTEND or NOEXTEND as the first statement in a program to set the mode for the current program.

The following examples show use of EXTEND as a command and a statement. These examples assume you are working on a system where NOEXTEND is the default mode.

Here the EXTEND command puts you in EXTEND mode:

```
EXTEND

Ready
```

You stay in EXTEND mode until you either:

- Enter the NOEXTEND command
- Use the NOEXTEND statement in your current program
- Run an executable (not a source) program
- Use a CCL or DCL command
- Switch to another keyboard monitor
- Log out

In the next example, you use EXTEND as a program statement:

```
NEW PROG1

Ready
100 EXTEND
```

You are in EXTEND mode while this program is your current program. However, the system puts you back in NOEXTEND mode if you run another program (with RUN, a CCL command, or a DCL command), create a program with NEW, or retrieve an existing program with OLD.

While BASIC–PLUS allows you to change modes within a program, this practice is not recommended. Write an entire program in either EXTEND or NOEXTEND mode – the program will be much easier to debug and maintain.

It is recommended that you work in EXTEND mode. You can use features not available in NOEXTEND mode, such as long variable names. In addition, programs written in EXTEND format are more compatible with other DIGITAL versions of BASIC. However, EXTEND format rules are more stringent than NOEXTEND format rules.

7.5.2 EXTEND and NOEXTEND Program Formats

Figures 7–2 and 7–3 show the differences between EXTEND and NOEXTEND format. Figure 7–2 is line 300 of the program OCTDEC.BAS written in NOEXTEND format; Figure 7–3 is the same line written in EXTEND format.

Figure 7–2: NOEXTEND Format

```
300   O% = -1% \ D% = 0%
      \ FOR Z% = L% TO 1% STEP -1%
      \   O% = O% + 1%
      \   V% = ASCII (RIGHT(S$,Z%))
      \   IF V% < 48% OR V% > 55% THEN
                PRINT 'INVALID INPUT'
      \         GOTO200
        ! initialize order and accumulator for each digit
        ! from the low order to the high order,
        ! increment order,
        ! pick up next digit,
        ! if out of range: ask for more input
        ! and print message.
```

Figure 7–3: EXTEND Format

```
300   ORDER% = -1%                              ! initialize order &
      \ ACCUMULATOR% = 0%                       ! and accumulator &
      \ FOR Z% = LNGTH% TO 1% STEP -1%          ! for each digit from &
                                                ! the low order to &
                                                ! the high order &
      \ ORDER% = ORDER% + 1%                    ! increment order &
      \ VALUE% = ASCII(RIGHT(S$,Z%))            ! pick up next digit &
      \ IF VALUE% < 48% OR VALUE% > 55% THEN    ! if out of range: &
                  PRINT "INVALID INPUT"         ! print message &
      \           GOTO 200                      ! ask for more input &
```

EXTEND and NOEXTEND formats differ in their:

- Rules for naming variables and functions. (A function is a named set of instructions.)
- Use of spaces and tabs.
- Line continuation.
- Placement of comments.

7.5.2.1 Variable and Function Names

In EXTEND format, a variable or function name consists of a letter followed by 0 to 29 characters from the set:

Letters	(A,B,...,Z and a,b,...,z)
Digits	(0,1,...,9)
Period	(.)

Some examples are RECORD.NUMBER, BIG47, J.10. If you use an existing BASIC–PLUS keyword or built-in function name, BASIC–PLUS returns the error "%Illegal symbol." (Appendix A lists all BASIC–PLUS keywords.)

In NOEXTEND format a variable or function name consists of a letter optionally followed by a digit. Some examples are A, A1, B4.

Note that in both EXTEND and NOEXTEND formats, you add a percent (%) sign suffix to the variable name to indicate an integer variable; you add a dollar ($) sign suffix to the variable name to indicate a string variable. For example:

```
A%              RECORD.NUMBER%
A$              ADDRESS$
```

See Section 8.3.2 for more information.

7.5.2.2 Spaces and Tabs

Spaces and tabs are significant in EXTEND format but not in NOEXTEND format. Thus, the following statements are illegal in EXTEND mode:

`10GOTO200`	This statement needs a space between line number 10 and GOTO and between GOTO and line number 200. Write the statement as "10 GOTO 200".
`10 L ETX = Y*2 + 1`	The keyword LET contains embedded spaces that must be removed. In addition, a space must be inserted between the keyword LET and the variable name X. Write the statement as: "10 LET X = Y*2 + 1".

The next statement is valid in both EXTEND and NOEXTEND mode but has a different action in each:

```
10 LETX = Y*2 + 1
```

In EXTEND mode, this statement assigns the value of (Y*2 + 1) to the variable named LETX. In NOEXTEND mode this statement assigns the value of (Y*2 + 1) to the variable named X.

7.5.2.3 Continuation Lines

In EXTEND format you can continue a program line onto the next terminal or text line with either the LINE FEED key or the ampersand/RETURN key combination. The ampersand/RETURN key combination is the preferred method because:

- It is compatible with BASIC–PLUS–2
- It allows you to alternate comments and statements on a program line that continues over several text lines, as shown in Figure 7–3

In NOEXTEND format you must use the LINE FEED key to continue a program line onto the next terminal or text line.

7.5.2.4 Comments

When you write a program line on one terminal line, the rules for placing comments and statements on the same line are the same in EXTEND and NOEXTEND modes. You can place a comment after a statement, but you cannot place a statement after a comment.

When you continue a program line over two or more text lines, however, the rules for using comments differ depending on whether you indicate continuation with the ampersand/RETURN key combination, legal only in EXTEND mode, or the LINE FEED key, legal in both EXTEND and NOEXTEND modes.

You can alternate comments and statements on the same line, as shown in Figure 7–3, only if you use the ampersand/RETURN key combination (not the LINE FEED key) to indicate continuation. See Section 7.4 for details.

Chapter 8
Building Statements

Chapter 7 introduced you to BASIC–PLUS statements; this chapter describes the elements that statements are composed of: keywords, data, and expressions. In addition, this chapter summarizes the BASIC–PLUS character set, which you use to specify these elements.

8.1 BASIC–PLUS Character Set

BASIC–PLUS uses the full ASCII character set. This set includes:

- The letters A through Z, in both upper- and lowercase
- The digits 0 through 9
- Special characters, such as the period (.) and the asterisk (*)

Appendix D lists the complete BASIC–PLUS character set. The appendix includes special symbols and keys and lists ASCII character codes.

BASIC–PLUS treats upper- and lowercase letters the same except in string data and ignores characters in REMARK statements and comments.

8.2 Keywords

The BASIC–PLUS language reserves a set of words and phrases for its own use. These words and phrases are called keywords.

BASIC–PLUS keywords include:

- Statement names, such as PRINT and LET
- Built-in function names, such as SIN and TAN
- Built-in variable names, such as ERR and STATUS
- Option names, such as MODE and FILESIZE

You cannot use BASIC–PLUS keywords as names for your own variables or functions unless you modify them. For example, you cannot name a variable FILESIZE, but you can name it FILE.SIZE. Appendix A contains a complete list of BASIC–PLUS keywords.

8.3 Data

BASIC–PLUS has three types of data:

- Real data
- Integer data
- String data

Real data and integer data are numeric data; string data are character data. BASIC–PLUS stores the three types of data using different internal formats.

Real data are stored as floating-point numbers. (The term floating-point refers to the decimal point.) Unlike integers, which are whole numbers, real numbers can have a fractional part. You can specify a real number in decimal format or in exponential format. Both formats are explained in Section 8.3.1.1.

Floating-point numbers are stored in memory in either a two-word or four-word format, depending on which math package you have on your system. The two-word format, called single-precision, provides six significant digits; the four-word format, called double-precision, provides fifteen significant digits. Floating-point numbers, and thus real data, can range in value from 10 ^–38 to 10 ^38.

By default, BASIC–PLUS prints numbers with more than six digits in exponential or E format. BASIC–PLUS prints a minus sign after the E if the exponent is negative (1E–09) and prints a space after the E if the exponent is positive (1E 09). See Section 8.3.1.1 for a description of E format.

Integer data are whole numbers without decimal points. Integers are stored in memory in binary form. Each integer uses one word of memory. BASIC–PLUS converts the binary integer to a decimal number before returning its value to you. Integers can range in value from –32768 to 32767.

String data are stored as groups of ASCII characters. BASIC–PLUS treats all the characters in a string as a unit. Each character in a string uses one byte (half a word) of memory storage. BASIC–PLUS keeps track of the length of a string as well as the characters it contains.

Table 8–1 summarizes the three data types.

Table 8–1: BASIC–PLUS Data Types

Data Type	Internal Format	Value Range
Real	Two-word or four-word floating-point	10 ^–38 to 10 ^38
Integer	One-word binary	–32768 to 32767
String	ASCII format; one byte (half-word) per character	0 to 32767 characters per string

All three types of data can be represented as constants or variables. Constants keep the same value throughout a program and are specified in the program itself. Variables can change in value during program execution. When you write expressions, you use constants and variables as operands. The following sections describe constants and variables in more detail.

8.3.1 Constants

BASIC–PLUS has real, integer, and string constants.

8.3.1.1 Real Constants

A real constant is one or more decimal digits, either positive or negative, with an optional decimal point. For example:

```
+2.
-3.675
1234.56
-123456
-.000001
```

These numbers are in decimal format. Although the decimal point is optional, its use is recommended because it avoids unnecessary data conversions and makes your program easier to read.

You can input large and small numbers using exponential format. This mathematical shorthand uses the format:

$$\left\{\begin{matrix}+\\-\end{matrix}\right\} \text{number E} \left\{\begin{matrix}+\\-\end{matrix}\right\} \text{n}$$

where:

+ or –	is the number's sign. The plus sign is optional, but the minus sign for a negative number is required.
number	is the number carried to a maximum of six decimal places for single-precision or fifteen decimal places for double-precision.

E	stands for "multiplied by 10 to the power of."
+ or –	is the exponent's sign. The plus sign is optional, but the minus sign is required.
n	is an integer constant (the power of 10). It can be zero but not blank.

For example:

```
.000123456          can be written as   123456E-9
1234560000.         can be written as   123456E4
-12345678900.       can be written as   -1.23456789E10
```

The E format is flexible; you can write .001 as 1E–3, .01E–1, or 100E–5.

Note that, by default, BASIC–PLUS prints numbers with fewer than six digits in decimal format and numbers with more than six digits in exponential format.

8.3.1.2 Integer Constants

An integer constant is a whole number followed by a percent sign. For example:

```
29%         -8%
3432%       1%
12345%      205%
```

Integer constants can range in value from –32767% to 32767%. Because the largest integer has only five digits, BASIC–PLUS always prints integers in decimal format.

8.3.1.3 String Constants

A string constant is a series of ASCII characters enclosed in single or double quotation marks (called the string delimiter). Some examples of string constants are:

```
"THE RECORD NUMBER DOES NOT EXIST"
'THIS PROGRAM GIVES RANDOM DICE ROLLS'
"4 Baymeadow Drive"
```

Characters in string constants can be letters, numbers, spaces, tabs, or any ASCII character except the string delimiter. BASIC–PLUS prints every character between delimiters exactly as you type it in the source program, including upper- and lowercase letters; tabs; special characters; and leading, trailing, and embedded spaces. For example, these two string constants are different:

```
"END-OF-FILE REACHED"
"     END-OF-FILE REACHED       "
```

BASIC–PLUS does not print the delimiting quotation marks when executing the program. For example:

```
10 PRINT "END-OF-FILE REACHED"
20 END
RUNNH
END-OF-FILE REACHED
```

To print quotation marks, enclose single quotation marks inside double marks or vice versa. For example:

```
10 PRINT 'FAILURE CONDITION:  "RECORD LENGTH"'
20 END
RUNNH
FAILURE CONDITION:  "RECORD LENGTH"
```

You cannot mix single and double quotation marks. The following is not a valid string:

```
"Quotation marks do not match'
```

BASIC–PLUS lets you omit the second quotation mark but not the first. This string is valid:

```
"Second quotation mark missing
```

But this string is not valid:

```
First quotation mark missing"
```

A string can contain up to 32767 characters but is limited by the amount of available memory. You cannot use the LINE FEED key or the &/RETURN key combination to type a string on two or more terminal lines. To create a string longer than a terminal line, use string concatenation, described in Section 8.4.2, or other string operations described in Chapter 10. You can also save long strings in disk files.

8.3.2 Variables

You specify constants as numbers or quoted strings; you specify variables by name. A variable reserves a location in the computer's memory. The variable names the location and lets you assign any legal number or string to it. Each memory location can hold only one value at a time.

Depending on the operations you specify, the value of a variable can change as each statement in your program is executed. BASIC–PLUS uses the most recently assigned value when performing calculations. The value remains the same until another statement assigns a new value to the variable.

BASIC–PLUS sets all numeric variables equal to 0 and all string variables equal to the null (empty) string before program execution. Thus, you need to assign an initial value to a variable only when you want a different value. However, it is always good programming practice to assign initial values to variables at the beginning of the program.

BASIC–PLUS has real, integer, and string variables. The two types of numeric variables, real and integer, reserve locations for numeric values. String variables reserve locations for string values. All types of variables can have subscripts. Subscripted variables break up memory into compartments and let you store data in arrays. (See Chapter 9.)

8.3.2.1 Naming Variables

All variable names must start with a letter and can be followed by up to 29 letters, digits, or periods (in EXTEND mode) or a single digit (in NOEXTEND mode). Variable names cannot contain embedded spaces.

You add a percent sign (%) suffix to the variable name to define an integer variable; you add a dollar sign ($) suffix to the variable name to define a string variable. Real variable names have no suffix.

Note that variables with the same name but different suffixes are different variables. For example, if you use A and A% in the same program, BASIC–PLUS defines two variables: A, a real variable, and A%, an integer variable. The next three sections describe each type of variable.

8.3.2.2 Real Variables

A real variable is a named memory location that stores a real (floating-point) number. For example:

```
C         L.,.5       ID.NUMBER
M1        BIG47       STORAGE.LOCATION.FOR.XX
```

If you assign an integer constant to a real variable, BASIC–PLUS converts the integer to a real number. For example:

```
10 LET A = 10%
20 PRINT A
30 END
RUNNH
 10
```

8.3.2.3 Integer Variables

An integer variable is a named memory location that stores a whole number. An integer variable name ends with a percent sign (%). For example:

```
B%        C.8%        RECORD.NUMBER%
A3%       D6E7%       COUNTER%
```

If you assign a real constant to an integer variable, BASIC–PLUS drops the fractional portion of the constant; it does not round to the nearest integer. For example:

```
B% = 5.7
```

BASIC–PLUS assigns the value 5 to the integer variable, not 6.

Use integer variables, not real variables, for working with whole numbers in the range –32767 to 32767. Integers use less memory storage space than floating-point numbers and are more efficient for the computer to process.

8.3.2.4 String Variables

A string variable is a named memory location that stores a string value. A string variable name ends with a dollar sign ($). For example:

```
A$          M2$          EMPLOYEE.NAME$
T..$        L.6$         TEXT$
```

Strings have both value and length. BASIC–PLUS sets all string variables to a default length of zero when it translates your program. During program execution, however, the length of a character string associated with a string variable can vary from zero (a null or empty string) to 32767 characters (including spaces).

8.4 Expressions

Expressions consist of operands (constants, variables, functions) separated by arithmetic, relational, or logical operators. Depending on the data types of the operands (real, integer, string), the three types of operators produce:

- Arithmetic expressions
- String expressions
- Relational expressions
- Logical expressions

Here are some examples of expressions:

```
PI * RADIUS ^2                  Arithmetic Expression
FILENAME$ + "." + TYPE$         String Expression
A < B                           Relational Expression
(A < B) AND (B < C)             Logical Expression
```

Like a constant and a variable, an expression is a way to represent a value in BASIC–PLUS. When you use an expression in a statement, BASIC–PLUS evaluates or "solves" it, using the same kinds of rules you do when you solve mathematical problems. After evaluating the expression, BASIC–PLUS uses the result in program execution.

In many BASIC–PLUS statements, you can specify either a constant, a variable, a function, or an expression. (Functions are described later in this manual.) The KILL statement, which deletes a file, is a good example. In Chapter 5, you learned how to use KILL as an immediate mode statement; KILL can also be used in a program.

When you write a KILL statement, you specify the file to be deleted as a string. The string can be a string constant, a string variable, a string function (see Chapter 10), or a string expression. For example:

```
10 KILL "TAXES.DAT"           String Constant
20 KILL FILESPEC$             String Variable
30 KILL FILENAME$ + ".DAT"    String Expression
```

The string constant in line 10 specifies the file directly; BASIC–PLUS deletes TAXES.DAT every time this KILL statement is executed.

The string variable in line 20 refers to a location where a file specification is stored. When this KILL statement is executed, BASIC–PLUS deletes the file whose file specification is currently stored in FILESPEC$. Depending on how the rest of the program is written, the contents of FILESPEC$ may differ each time the program is run or each time this KILL statement is executed during the same program run.

The string expression in line 30 contains both a string variable and a string constant. When this KILL statement is executed, BASIC–PLUS evaluates the string expression to determine which file to delete. The plus (+) sign causes BASIC–PLUS to form a single string made up of the characters stored in the variable FILENAME$ followed by the characters ".DAT". BASIC–PLUS deletes the file specified by this string. Like line 20, this expression may cause a different file to be deleted each time the program is run or each time this KILL statement is executed during the same program run.

Many statement descriptions in this manual indicate that you must supply an expression. When you write these statements, you can use a constant, a variable, a function, or an expression.

The following sections describe the four types of expressions in detail and explain how BASIC–PLUS evaluates them.

8.4.1 Arithmetic Expressions

Arithmetic expressions are real or integer operands separated by arithmetic operators. These expressions tell BASIC–PLUS to add, subtract, multiply, or divide numbers or to raise a number to a power, which is called exponentiation. The five arithmetic operators are shown in Table 8–2.

Besides the circumflex (^), you can also use double asterisks (**) to denote exponentiation. However, the circumflex is preferred because it is the standard BASIC–PLUS symbol for exponentiation. Use of the circumflex ensures compatibility with other versions of BASIC.

Table 8–2: Arithmetic Operators

Operator	Example	Meaning
+	A+B	Adds B to A
–	A–B	Subtracts B from A
*	A*B	Multiplies A by B
/	A/B	Divides A by B
^	A^B	Calculates A to the B power

The following examples show how BASIC–PLUS handles real and integer data in arithmetic expressions. An operation on two numeric operands of the same data type yields a result of that type. For example:

A% + B% produces an integer value.

A + B produces a real value.

An integer and a real quantity produce a real value. For example:

A * B% produces a real value.

6.87 * 5% produces a real value.

When you assign a value of one data type to a variable of a different data type, BASIC–PLUS converts the value to the variable's data type. For example:

```
A = 5% * 35%
```

Even though 5% and 35% are integer values, this statement assigns the real value 175.0 to the real variable A.

In general, two arithmetic operators cannot occur consecutively in the same expression. The exceptions are the unary plus and unary minus, which specify that a number is positive or negative. For example:

A * +B is valid

A * –B is valid

A * (–B) is valid

A – * B is not valid

Because BASIC–PLUS assumes unsigned numbers are positive, the unary plus is optional. You must use the unary minus to specify a negative number.

8.4.2 String Expressions

String expressions are strings separated by the plus sign (+). The plus sign concatenates (combines) the strings. For example:

```
10 C$ = "The street name" + " is Seashore Drive"
20 PRINT C$
30 END
RUNNH
The street name is Seashore Drive
```

You can use string expressions to:

- Create long strings
- Combine string constants and string variables
- Combine several string variables

8.4.3 Relational Expressions

A relational expression consists of two operands (constants, variables, functions, or expressions) separated by a relational operator. The relational operator causes BASIC–PLUS to compare the two operands and determine if the indicated relationship is true or false.

There are two types of relational expressions, numeric and string. Numeric relational expressions compare numeric operands; string relational expressions compare string operands.

You use relational expressions in conditional statements (such as IF–THEN statements and conditional FOR loops) and in statement modifiers to control the order of program execution. String relational expressions also let you sort string data into alphabetical order.

The following IF–THEN statement contains a relational expression that controls program execution:

```
300 IF ANSWER$ = "YES" THEN GOTO 1000
```

When this statement is executed, BASIC–PLUS tests the current value of the string variable ANSWER$. If ANSWER$ contains the value YES, BASIC–PLUS evaluates the expression ANSWER$ = "YES" as "true" and control goes to line 1000 in the program. If ANSWER$ contains any other string value, BASIC–PLUS evaluates the expression ANSWER$ = "YES" as "false" and control goes to the next line number in the program.

Relational expressions are evaluated as –1 if the relation is true and 0 if the relation is false. These numbers are *logical values*. You need not remember these values to use relational expressions; BASIC–PLUS uses them to mean "true" and "false" and continues executing your program accordingly. You do need to know these values, however, if you want to process them in a program. (Chapter 11 explains how to process logical values.)

The rest of this section describes relational expressions in detail and gives more examples of how to use them.

8.4.3.1 Numeric Relational Expressions

Numeric relational expressions consist of numeric operands separated by relational operators. When used with numeric operands, relational operators compare the operands to determine if a numeric expression is true or false.

Table 8–3 lists numeric relational operators.

Table 8–3: Numeric Relational Operators

BASIC–PLUS Symbol	Example	Meaning
=	A=B	A is equal to B.
<	A<B	A is less than B.
<=	A<=B	A is less than or equal to B.
>	A>B	A is greater than B.
>=	A>=B	A is greater than or equal to B.
<>	A<>B	A is not equal to B.
==	A==B	A is approximately equal to B.

The term "approximately equal to" means that the two quantities look the same when you print them with the PRINT statement. Numbers are stored internally at greater than 6 digits of precision but are rounded to 6 digits for output. Two numbers that are identical when rounded to 6 digits of precision are approximately equal (==). These numbers may be close enough in value for some calculations. By contrast, two numbers equal to the internal limits of precision are truly equal (=).

The next example compares two real numbers:

```
10 INPUT "Enter a value for A";A
20 INPUT "Enter a value for B";B
30 IF A = B THEN PRINT "A is equal to B"
40 IF A < B THEN PRINT "A is less than B"
50 IF A > B THEN PRINT "A is greater than B"
32767 END
```

When you run this program, it prompts you to enter values for the real variables A and B. It compares the two values and then prints one of three messages depending on the values you entered. For example:

```
RUNNH
Enter a value for A? 10.02RET
Enter a value for B? 10.021RET
A is less than B

Ready
```

8.4.3.2 String Relational Expressions

String relational expressions consist of string operands separated by relational operators. The relational operators compare the value of the string operands, which can be strings or string expressions. Like numeric relational expressions, string relational expressions are evaluated as true or false.

BASIC–PLUS uses the ASCII character collating sequence to determine the relative character values (see Appendix D). It compares the strings character by character, from left to right, until it finds a difference in ASCII value. Because the ASCII collating sequence is an alphabetic sequence, you can use string relational expressions to sort string data into alphabetical order.

Table 8–4 lists string relational operators and their meanings.

Table 8–4: String Relational Operators

Operator	Example	Meaning
=	A$ = B$	The strings A$ and B$ are equal, exept for possible trailing spaces.
<	A$ < B$	String A$ precedes string B$ in alphabetic sequence.
<=	A$ <= B$	String A$ is equal to or precedes string B$ in alphabetic sequence.
>	A$ > B$	String A$ follows string B$ in alphabetic sequence.
>=	A$ >= B$	String A$ is equal to or follows string B$ in alphabetic sequence.
<>	A$ <> B$	The strings A$ and B$ are not equal.
==	A$ == B$	The strings A$ and B$ are identical, including trailing spaces. That is, the strings have both the same length and character composition.

BASIC–PLUS ignores trailing spaces in a string comparison, except when you use the == operator. When you use other operators, "YES" is the same as "YES ".

Like numeric relational expressions, string relational expressions let you control the order in which statements are executed. For example:

```
10 A$ = "ABC"
20 B$ = "DEF"
30 IF A$ < B$ GOTO 60
40 PRINT A$
50 PRINT B$
60 END
```

When you run this program, BASIC–PLUS compares the first character in each string to determine if A$ occurs first in ASCII collating sequence. Because "A" precedes "D" in the ASCII table, A$ precedes B$ in the collating sequence. Program control shifts to line 60 and program execution ends. No values are printed.

If you change the program so that A$ is the string "DEF" and B$ is the string "ABC", A$ no longer precedes B$ in the collating sequence. Program execution continues at line 40. The program prints the values of A$ and B$ before execution ends. For example:

```
10 A$ = "DEF"
20 B$ = "ABC"
30 IF A$ < B$ GOTO 60
40 PRINT A$
50 PRINT B$
60 END
RUNNH
DEF
ABC
```

When BASIC–PLUS compares two strings of unequal length, it compares the shorter string (length n) with the first n characters of the longer string. When the first n characters of the strings are the same, the strings are equal if the rest of the characters in the longer string are blanks. Otherwise, the longer string is greater than the shorter string. The following example compares strings of equal and unequal lengths:

```
100 INPUT "Enter a value for A$";A$
200 INPUT "Enter a value for B$";B$
300 IF A$ > B$ THEN &
        PRINT A$; " COMES AFTER " ; B$ &
    ELSE IF B$ > A$ THEN &
        PRINT A$; " COMES BEFORE " ; B$ &
    ELSE PRINT A$; " IS THE SAME AS "; B$
32767 END
```

When you run this program, it prompts you to enter values for the string variables A$ and B$. It compares the two values and then prints one of three messages depending on the values you entered. For example:

```
RUNNH
Enter a value for A$? ABC<RET>
Enter a value for B$? ABCDEF<RET>
ABC COMES BEFORE ABCDEF

Ready

RUNNH
Enter a value for A$? ABCDEF<RET>
Enter a value for B$? ABC<RET>
ABCDEF COMES AFTER ABC

Ready
```

```
RUNNH
Enter a value for A$? APPLE      RET
Enter a value for B$? APPLERET
APPLE      IS THE SAME AS APPLE

Ready
```

A null string has a length of zero and is therefore less than any string of length greater than zero. However, a null string is the same as a string of blanks. Consider the following immediate mode example (the symbol "" represents a null string):

```
IF "" = "    " THEN PRINT "EQUAL" ELSE PRINT "NOT EQUAL"
EQUAL
```

8.4.4 Logical Expressions

A logical expression contains either:

- One operand and a logical operator
- Two operands separated by a logical operator

Here are some examples of logical expressions:

```
(A < 0.) AND (B = 1.)
((A > B) OR (C > D)) AND A/B <> C/D
NOT A
```

Logical expressions are valid wherever numeric expressions are valid.

There are two types of logical expressions: those with relational expressions as operands and those with integers as operands. The two types of logical expressions have different uses in BASIC–PLUS programming. This section describes logical expressions with relational expressions as operands; see Chapter 11 for information on logical expressions with integers as operands.

Logical expressions that have relational expressions as operands are like relational expressions. BASIC–PLUS tests the relationship between the two operands and evaluates the expression as true or false. The logical operator determines what type of test BASIC–PLUS performs (see Table 8–5). Like relational expressions, this type of logical expression returns –1 for true and 0 for false.

You use this type of logical expression the same way you use relational expressions; that is, in IF–THEN statements, conditional FOR loops, and certain other statements to control the order in which program statements are executed.

Table 8–5 lists the logical operators. Assume that A and B are relational expressions.

Table 8–5: Logical Operators

Operator	Example	Meaning
NOT	NOT A	The logical opposite of A. If A is true, NOT A is false. If A is false, NOT A is true.
AND	A AND B	The logical product of A and B. A AND B is true only if A is true and B is true.
OR	A OR B	The logical sum of A and B. A OR B is true if either A or B or both are true. A OR B is false only if both A and B are false.
XOR	A XOR B	The logical exclusive OR of A and B. A XOR B is true if either A or B (but not both) is true. Otherwise, A XOR B is false.
EQV	A EQV B	The logical equivalence of A and B. A EQV B is true if A and B are both true or both false. Otherwise, A EQV B is false.
IMP	A IMP B	The logical implication of A and B. A IMP B is false if A is true and B is false. Otherwise, A IMP B is true.

The truth values in Table 8–6 summarize the results of these logical operations.

Table 8–6: Truth Values for Logical Operations

A	B	A AND B
T	T	T
T	F	F
F	T	F
F	F	F

A	B	A OR B
T	T	T
T	F	T
F	T	T
F	F	F

A	B	A XOR B
T	T	F
T	F	T
F	T	T
F	F	F

A	B	A EQV B
T	T	T
T	F	F
F	T	F
F	F	T

A	B	A IMP B
T	T	T
T	F	F
F	T	T
F	F	T

A	NOT A
T	F
F	T

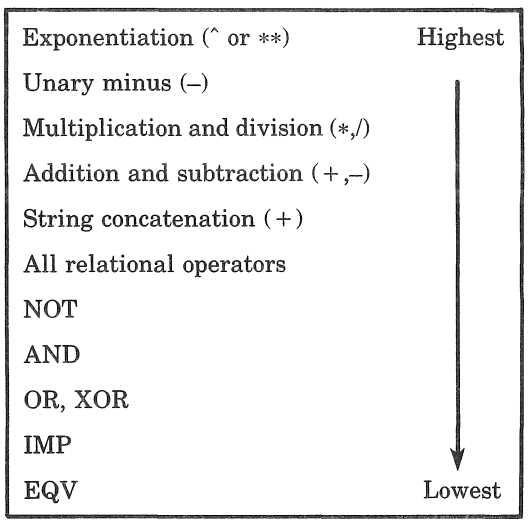
Note the following:

- The operators XOR and EQV are opposites.
- BASIC–PLUS generally accepts any nonzero value as true.

8.4.5 How Expressions Are Evaluated

BASIC–PLUS evaluates expressions according to its rules of operator precedence. Imagine a list of all the arithmetic, string, relational, and logical operators described in this chapter. Each operator has a fixed position or rank in this list. The operator's position tells BASIC–PLUS when to perform the operation. You can use parentheses to change the order of precedence. Table 8–7 shows operator precedence.

Table 8–7: Operator Precedence



BASIC–PLUS evaluates expressions according to the following rules:

1. Expressions inside parentheses are evaluated first. When you place one set of parentheses inside another set (called nesting), the expressions are evaluated from the inside out. For example:

```
B = 5 * (2 ^ (3 + 2))
```

   This expression assigns the value 160 to B. Because (3 + 2) is the innermost parenthetical expression, BASIC–PLUS evaluates it first. Then it evaluates (2 ^ 5) and finally (5 * 32).

2. After expressions in parentheses are evaluated, BASIC–PLUS performs operations in the order they appear in Table 8–7. The expression from the previous example without parentheses shows the order of operations:

```
B = 5 * 2 ^ 3 + 2
```

This expression assigns the value 42 to B. First BASIC–PLUS evaluates 2 ^ 3, then 5 * 8, and finally 40 + 2.

3. When expressions contain operators of equal rank in the table, BASIC–PLUS performs operations from left to right. For example, BASIC–PLUS evaluates A^B^C as (A^B)^C, A/B/C as (A/B)/C, and A*B/C as (A*B)/C.

You are encouraged to use parentheses even though they are not required. They make expressions easier to read and also help you write correct expressions.

Chapter 9
Elementary Statements and Features

This chapter describes the elementary BASIC–PLUS statements and features. Here is a guide to its contents:

Topic	Statements or Functions
Assigning Values to Variables	LET
Introduction to Input/Output	PRINT, INPUT, READ, DATA, RESTORE
Unconditional Branching	GOTO
Conditional Branching	IF–THEN, IF–GOTO
Program Loops	FOR–NEXT, WHILE–NEXT, UNTIL–NEXT
Subscripted Variables	DIM
Mathematical Functions	ABS, SGN, INT, FIX, COS, SIN, TAN, ATN, SQR, EXP, LOG, LOG10, PI, RND, RANDOMIZE
User-Defined Functions	DEF*
Subroutines	GOSUB, RETURN
Ending or Halting Execution	END, STOP

The statements and features described in this chapter are sufficient, by themselves, for the solution of many programming problems. Once you master them, you can investigate their more advanced applications, as well as other statements and features described in Parts III and IV. The more advanced features can help you solve more complex problems and write more efficient programs.

9.1 LET Statement

The LET statement assigns a value to a variable. LET has the form:

[LET] <variable(s)> = <expression>

The LET statement does not indicate algebraic equality. Instead, it assigns the results of the expression to the indicated variable. For example:

```
10 LET X = X+1
20 LET W2 = (A4-X3)*(Z-A/B)
```

In line 10, the old value of X is increased by 1 and becomes the new value of X. In line 20, the expression on the right-hand side is evaluated using the current values for A4, X3, Z, A, and B, and the result is assigned to W2.

The LET statement can be a simple numerical assignment, such as:

```
50 LET A = 35
```

For example:

```
50 LET A = 35
60 PRINT A
70 LET A = A+1
80 PRINT A
90 END
RUNNH
 35
 36

Ready
```

The LET statement can also evaluate an expression that is continued on more than one line. For example:

```
40 LET X = (W-X4*3)*(Z-A/ &
   (A-B)-17)
```

For convenience, you can omit the word LET from the LET statement. Omitting LET does not change the effect of the statement. For example, you may find it easier to type the first statement than the second statement:

```
10 X = 12*(S+7)

10 LET X = 12*(S+7)
```

You can use the LET statement anywhere in a multiple statement line. For example:

```
10 X = 44 \ Y =X^2+Y1 \ B2 = 35*A
```

The LET statement also lets you assign the same value to several variables in the same statement. For example:

```
10 LET X, Y, Z = 5.7
```

Line 10 sets each of the three variables equal to 5.7.

Do not use a LET statement of this form to assign the same value to several variables:

```
20 LET X = Y = 5.7
```

This statement assigns the value –1 to X if the expression (Y = 5.7) is true and the value 0 to X if the expression (Y = 5.7) is false. (See Section 8.4.3.)

9.2 Introduction to Programmed Input and Output

This section describes the basic techniques for performing BASIC–PLUS program input/output, usually abbreviated to I/O. The term I/O means bringing data from a file or device into a program for processing and sending it to a file or device after processing. The file may be stored on disk or magnetic tape; the device may be a terminal, a line printer, or other peripheral device.

You perform basic I/O operations in BASIC–PLUS with PRINT and INPUT statements. The basic forms of these statements are presented in this section to help you create simple BASIC–PLUS programs and get tangible results. This section also describes the READ, DATA, and RESTORE statements, which you use to supply a fixed list of values to your program. The values are contained in the program itself; they are not supplied from outside the program.

Part IV of this manual describes more advanced I/O techniques.

9.2.1 PRINT Statement

The PRINT statement prints data to a device or file. In this introduction to the PRINT statement, your terminal is the output device. Chapter 15 contains a complete description of the PRINT statement.

The format of the PRINT statement is:

PRINT [list]

The list, which is optional, can contain expressions, character strings, numeric constants, and variables. A PRINT statement without any operands prints a blank line.

PRINT statements evaluate expressions and print results. All expressions in a list are evaluated before BASIC–PLUS prints a value. Consider the following program:

```
LISTNH
2000    LET A=1 \ LET B=2 \ LET C=3+A
2100    PRINT
2200    PRINT A+B+C
32767   END

Ready

RUNNH

 7

Ready
```

You can use the PRINT statement anywhere in a multi-statement line. For example:

```
10 A=1 \ PRINT A \ A=A+5 \ PRINT \ PRINT A
```

This line prints:

```
 1

 6
```

BASIC–PLUS prints a carriage return/line feed at the end of each PRINT statement by default. Thus, the first PRINT statement prints a "1" and a carriage return/line feed. The second PRINT statement prints the blank line, and the third PRINT statement prints a "6" and another carriage return/line feed.

9.2.1.1 Printing Numbers and Character Strings

When printing positive numbers, BASIC–PLUS adds a leading and a trailing blank space. When printing negative numbers, BASIC–PLUS adds a leading minus sign and a trailing blank space. For example:

```
PRINT 10;-20;30
 10 -20  30
```

(The semicolon prints values in a "packed" format – see Section 9.2.1.2.)

The PRINT statement can also print characters. You delimit characters for printing by placing single or double quotation marks at each end of the string. The same type of quotation mark is necessary at the beginning and the end of each string. For example:

```
LISTNH
100     PRINT "TIME'S UP"
110     PRINT 'QUOTH THE RAVEN, "NEVERMORE"'
32767   END

Ready

RUNNH
TIME'S UP
QUOTH THE RAVEN, "NEVERMORE"

Ready
```

When BASIC–PLUS prints a string, it does not add leading or trailing spaces. Only the characters between quotation marks appear. To add leading and trailing spaces, type them inside the quotation marks with the keyboard space bar. Spaces are output exactly as you type them inside the quotation marks.

You can also use PRINT to print combinations of characters and numeric values. For example:

```
550 X=83.4
580 PRINT "AVERAGE GRADE IS";X
```

This example prints:

```
AVERAGE GRADE IS 83.4
```

9.2.1.2 Formatting the Output

You specify how you want output formatted by using either commas or semicolons between items in the PRINT list. Use commas if you want values widely spaced across the terminal line; use semicolons if you want values printed without added spaces.

Commas print items in print zones. BASIC–PLUS considers a terminal line to be divided into print zones of 14 spaces each. On terminals where the maximum print line contains 80 character positions, there are 5 print zones. Terminals with 84 or more characters per line have additional print zones of 14 spaces each.

When a comma follows an item in a PRINT statement, the next value to be printed appears in the next available print zone. For example:

```
LISTNH
10 LET A=3, \ LET B=2,
20 PRINT A,B,A+B,A*B,A-B,B-A,A^B

Ready

RUNNH

 3             2             5             6             1
-1             9
```

BASIC–PLUS prints the sixth element in the PRINT list as the first entry on a new line because an 80–character line has five print zones.

Two commas together in a PRINT statement cause a print zone to be skipped. For example:

```
LISTNH
100     REM       ILLUSTRATES PRINT ZONES AND LINE CONTINUATIONS.
110     LET A=1 &
        \ LET B=2 &
        \ PRINT A,B,,A+B &
        !NOTE DOUBLED COMMA AFTER B
32767   END

Ready

RUNNH
 1           2                       3

Ready
```

Use the semicolon between PRINT list items instead of the comma if you want a tighter packing of printed values. A semicolon prints no extra spaces. The following examples compare the effects of the semicolon and comma.

```
LISTNH
100      LET A=1 &
         \ LET B=2 &
         \ LET C=3
110      PRINT A;B;C;
120      PRINT A+1; B+1; C+1
130      PRINT A,B,C,
140      PRINT A+B+C,C^B,C^(B^C)
150      PRINT
160      PRINT 50,100;150;200,250,300
32767    END

Ready

RUNNH
 1  2  3  2  3  4
 1             2             3             6             9
 6561

 50            100  150  200               250           300

Ready
```

```
LISTNH
150     LET X=2574.          !STUDENT NUMBER
160     LET G=89.            !GRADE
170     LET A=90.6           !AVERAGE
180     LET R=14             !RANK

              .
              .
              .

300     N=562
640     PRINT 'STUDENT NUMBER'; X, 'GRADE ='; G;
650     PRINT 'AVERAGE ='; A
660     PRINT '     RANK IN CLASS'; R; 'OF'; N
32767   END

Ready

RUNNH
STUDENT NUMBER 2574         GRADE = 89 AVERAGE = 90.6
     RANK IN CLASS 14 OF 562

Ready
```

You can omit the semicolon between a text string and another item. However, its use is recommended for compatibility with BASIC–PLUS–2.

As noted, BASIC–PLUS automatically prints a carriage return/line feed at the end of each PRINT statement. To suppress the automatic carriage return/line feed, end the PRINT list with a comma or a semicolon. If you use a comma, the next PRINT statement starts printing in the next available print zone. For example:

```
LISTNH
110     LET A = 1 &
        \   B = 2 &
        \   C = 3
130     PRINT A,
140     PRINT B
150     PRINT C
32767   END

Ready

RUNNH
 1             2
 3

Ready
```

If you use a semicolon, the next PRINT statement starts printing in the next space on the terminal line.

9.2.2 INPUT Statement

The INPUT statement provides data to a program while it is running. The data comes from a device or a file. In this section, which introduces the INPUT statement, your terminal is the input device. Chapter 15 of this manual contains a complete description of the INPUT statement.

The INPUT statement has the form:

INPUT <variable list>

The list can contain real, integer, and string variables. When an INPUT statement is executed, the program halts, and BASIC–PLUS prints a question mark on the terminal and waits for you to supply data.

For example, suppose you enter this INPUT statement into a program:

```
50 INPUT A,B,C
```

During program execution, this statement causes BASIC–PLUS to print a question mark on the terminal and wait for you to enter three numeric values. For example:

```
RUNNH
?
```

In response to the question mark, type three numeric values separated by commas. After entering the values, press the RETURN key to pass the values to your program.

```
? 5,6,7RET
```

If you type too few values on a line, BASIC–PLUS prints another question mark to indicate that it needs more data. If you type too many values on a line, BASIC–PLUS ignores any extra ones. Enter no more than 132 characters, including commas between values, before you press the RETURN key. The INPUT statement allows you to enter more than 132 characters on a line but may not always read long input lines correctly.

The example just shown points out a problem with the prompting question mark; it lets you know that input is expected, but it does not tell you what kind of input to enter or how many values to type. To avoid this problem, you can add a PRINT statement to print a prompting message at execution time. For example:

```
40 PRINT "Enter 3 numbers separated by commas and press RETURN."
50 INPUT A,B,C

RUNNH
Enter 3 numbers separated by commas and press RETURN.
?
```

You can also include a prompting message in the INPUT statement itself. Use single or double quotation marks to set the message off from other parts of the statement. For example, the following statements have the same result:

```
110 INPUT "YOUR AGE IS";A

110 INPUT 'YOUR AGE IS';A
```

Either statement has the same result as:

```
110 PRINT "YOUR AGE IS";
120 INPUT A
```

When you include a prompting message in an INPUT statement, you can use either a comma or a semicolon to separate the string you want printed from the input variable names. These formatting characters work the same way they do in the PRINT statement.

The following program computes the academic standing of a student. Four INPUT statements prompt for the necessary data. Each INPUT statement prints a prompt to indicate what kind of data to enter. The program uses PRINT statements to print the results. The PRINT statements also print text to describe the results. The output for the program begins after the command RUNNH.

```
LISTNH
150       INPUT 'STUDENT NUMBER';X
160       INPUT 'GRADE';G
170       INPUT 'AVERAGE';A
180       INPUT 'RANK';R

                .
                .
                .

300       N=562
640       PRINT 'STUDENT NUMBER'; X, 'GRADE ='';G;
650       PRINT 'AVERAGE ='; A
660       PRINT '     RANK IN CLASS'; R; 'OF'; N
32767     END

Ready

RUNNH
STUDENT NUMBER? 2574
GRADE? 89
AVERAGE? 90.6
RANK? 14
STUDENT NUMBER 2574          GRADE = 89 AVERAGE = 90.6
     RANK IN CLASS 14 OF 562

Ready
```

9.2.3 READ and DATA Statements

Like the INPUT statement, the READ and DATA statements supply data to a program. However, unlike INPUT, the data is contained in the program itself instead of coming from a source outside the program (such as your terminal).

READ and DATA differ from INPUT in another way. When you use INPUT to supply data, it can differ each time the program is run. On the other hand, READ and DATA supply a fixed list of data values to your program. You must edit the program to change the values.

A READ statement inputs the list of variables whose values it gets from a DATA statement. Neither statement works without the other, and the two statements must have compatible sequences.

A READ statement has the form:

READ <variable list>

A DATA statement has the form:

DATA <value list>

With a READ statement, the variables you list are assigned values sequentially from the set of DATA statements in the program. Before you run the program, BASIC–PLUS takes all DATA statements in the order they appear and creates a data block. Each time a READ statement is encountered, BASIC–PLUS supplies the next value from the data block. If the data block runs out of data and another READ statement is executed, BASIC–PLUS prints the message:

```
?Out of data at line n
```

READ and DATA statements appear as follows:

```
150 READ X, Y%, Z, S1, Y2, Q9
330 DATA 4, 2, 1.7
340 DATA 6.73E-3, -174.321, 3.1415927
```

Line 150 makes the assignments:

X = 4.0
Y% = 2%
Z = 1.7
S1 = 6.73E–3
Y2 = –174.321
Q9 = 3.1415927

Do not include the percent (%) character when you specify integer values in a DATA statement. If you do, you get the following error when you run the program:

```
%Data format error at line n
```

The next example contains real, integer, and string data:

```
10 READ A, A$, A%, B$
50 DATA 12.5, NOW, 12, "QUOTED STRING"
```

This example assigns 12.5 to the real variable A, NOW to the string variable A$, 12 to the integer variable A%, and QUOTED STRING to the string variable B$.

You can read a numeric value without a decimal point into an integer variable or a real variable. For example:

```
10 READ A                    10 READ A%
50 DATA 12                   50 DATA 12
```

In the first example, 12 is stored as a floating-point number. In the second example, 12 is stored as an integer. However, you cannot read a floating-point number into an integer variable:

```
10 READ A%
20 DATA 1.1
RUNNH
%Data format error at line 10
```

If you read a number into a string variable, the number is stored as a string of ASCII characters. For example:

```
30 READ A$
40 DATA 100
```

These statements read the character string "100" into A$.

When you specify strings in DATA statements, include quotation marks around strings that contain:

- One or more commas
- Significant spaces or tabs
- Lowercase letters that should not be converted to uppercase

If you do not include quotation marks, BASIC–PLUS ignores spaces and tabs in the string and converts lowercase letters to uppercase.

The optional MAT READ statement allows you to read matrices from DATA statements. See Chapter 12 for more information.

You must read in data before you can use it in a program. Thus, you normally place READ statements near the beginning of a program. You can place DATA statements anywhere in a program; however, it is good programming practice to place DATA statements together near the end of the program. DATA statements are read in order of their line numbers, so they must appear in the correct sequence.

You can place a READ statement anywhere in a multi-statement line. But a DATA statement must be the only statement in a line or the last statement in a multi-statement line.

You cannot place a comment at the end of a DATA statement. If you include a comment and the last item in the DATA statement is a string, BASIC–PLUS converts the comment to uppercase, removes spaces and tabs, and includes it in the string. If you include a comment and the last

item in the DATA statement is an integer or floating-point number, BASIC–PLUS returns the "%Data format error" when you run the program.

9.2.4 RESTORE Statement

You may need to use the same data more than once in a program. The RESTORE statement lets you recycle through the complete set of DATA statements in the program, beginning with the lowest numbered DATA statement.

The RESTORE statement has the form:

```
RESTORE
```

RESTORE causes the next READ statement to begin reading data from the first DATA statement in the program. It does so regardless of where it read the last data value.

You can use the same variable names the second time through the data, since the values are being read as though for the first time. Place dummy variables in the READ statement to skip unwanted values. (A "dummy" variable is a variable you make no further use of in the program.) Consider the following example of the RESTORE statement:

```
LISTNH
100    REM THIS PROGRAM ILLUSTRATES USE OF THE RESTORE STATEMENT
1500   READ N \ PRINT 'VALUES OF X ARE:'
1600   FOR I = 1 TO N \ READ X \ PRINT X,
1700   NEXT I
1800   RESTORE
1900   PRINT \ PRINT 'SECOND LIST OF X VALUES'
2000   PRINT 'FOLLOWING RESTORE STATEMENT:'
2100   FOR I=1 TO N \ READ X \ PRINT X,
2200   NEXT I
6000   DATA 4,1,2
6100   DATA 3,4
32767  END

Ready

RUNNH
VALUES OF X ARE:
 1            2            3            4
SECOND LIST OF X VALUES
FOLLOWING RESTORE STATEMENT:
 4            1            2            3

Ready
```

Note that BASIC–PLUS prints the following on the last line:

```
 4            1            2            3
```

These numbers are printed because BASIC–PLUS did not skip the value for the original N when it executed the loop beginning at line 2100.

9.3 Unconditional Branch, GOTO Statement

The GOTO statement transfers program execution immediately and unconditionally to a specified program line number. (Usually the specified line is not the next sequential line in the program.) The statement has the form:

GOTO <line number>

The line number to which program execution branches can be greater than, less than, or the same as the current line number.

Consider the following example:

```
LISTNH
10     LET A=2
20     GO TO 50
30     LET A=SQR (A+14)
50     PRINT A,A*A
32767  END

Ready

RUNNH
 2          4

Ready
```

When the program reaches line 20, control transfers to line 50. After line 50 is executed, the program ends. Line 30 is never executed. Any number of lines can be skipped in either direction.

Because any statement after GOTO on the same line is never executed, GOTO should always be the last statement on a multi-statement line. For example:

```
110     LET A=517, &
        \PRINT A &
        \GO TO 370
370     PRINT 'FINISHED'
32767   END
RUNNH
 517
FINISHED

Ready
```

```
110     LET A=517, &
        \GO TO 370 &
        \PRINT A
370     PRINT 'FINISHED'
32767   END
RUNNH
FINISHED

Ready
```

9.4 Conditional Branch, IF–THEN and IF–GOTO Statements

The IF–THEN and IF–GOTO statements conditionally transfer program execution to a specified line number or statement, depending on the outcome of some test or relationship. The format of the IF statement is:

IF <condition> { THEN <statement> | THEN <line number> | GOTO <line number> }

When the IF statement is executed, the specified condition is tested. If the condition is false, control goes to the next sequentially numbered line after the IF statement. If the condition is true, the statement after THEN is executed or control is transferred to the line number given after THEN or GOTO. (Section 13.5 describes an extension of this statement, the IF–THEN–ELSE statement.)

The deciding condition can be either:

- A simple relational expression where two numeric or string expressions are separated by a relational operator
- A logical expression where two relational or logical expressions are separated by a logical operator

For example:

Relational Expression	Logical Expression
`A + 2 > B`	`(A > B) AND (B <= SQR(C))`

When either type of expression is evaluated, the result is either true or false. Sections 8.4.3 and 8.4.4 describe the relational and logical operators. They also appear in Appendix A.

The following line contains a relational expression:

```
75 IF A*B>=B*(B+1) THEN LET D4=D4+1
```

This line compares the quantities A*B and B*(B + 1). If the first value is greater than or equal to the second value, the program increments the variable D4 by 1. The program does not increment D4 if B*(B + 1) is greater than A*B; instead, control passes immediately to the line following line 75.

When a line number follows the word THEN, the IF–THEN statement is the same as the IF–GOTO statement. Any executable BASIC–PLUS statement can follow the word THEN, including another IF statement. For example:

```
250 IF A>B THEN IF B>C THEN PRINT "A>B>C"
250 IF A>B AND B>C THEN PRINT "A>B>C"
```

The preceding two lines are logically equivalent. If B is less than A and greater than C, BASIC–PLUS prints:

```
A>B>C
```

Otherwise, the next line is executed.

In the following example, the IF–GOTO statement in line 110 limits the value of the variable A in line 100. It uses a *program loop* to do so (see Section 9.5). A program loop is a series of statements written so that, when the statements are executed, control transfers to the beginning of the statements. This process continues until some terminating condition is reached. Execution of the loop continues until the relationship A>4 is true, and then line 32767 is executed to end the program.

```
LISTNH
100       LET A=A+1 &
          \ X=A^2
110       IF A>4 GO TO 32767
120       PRINT 'X='X; ', AND VALUE OF A IS' A
130       GO TO 100
32767     END

Ready

RUNNH
X= 1  , AND VALUE OF A IS 1
X= 4  , AND VALUE OF A IS 2
X= 9  , AND VALUE OF A IS 3
X= 16  , AND VALUE OF A IS 4

Ready
```

The next example uses IF–THEN statements:

```
100 IF A>B THEN 340
200 IF (A=B) OR (B=C) THEN 260
300 IF A>B THEN A=-B                          ! CONDITIONAL ASSIGNMENT
400 IF (X>Y) IMP (Y>Z) THEN PRINT "QED"
```

To avoid confusion, you usually make an IF statement the last statement on a multi-statement line.

When an IF statement is not the last statement on a multi-statement line, all statements on a program line after the THEN keyword are executed only if the related expression is true.

For example:

```
LISTNH
90        INPUT 'ENTER A VALUE';A
100       IF A=1 THEN PRINT A; &
          \ PRINT "TRUE CASE" &
          \ GOTO 32767
110       PRINT "NOT = 1"
32767     END

Ready

RUNNH
ENTER A VALUE? 2
NOT = 1

Ready

RUNNH
ENTER A VALUE? 1
 1  TRUE CASE

Ready
```

9.5 Program Loops

Computers perform repetitive tasks with speed and accuracy. For example, a computer can compute the square roots of all integers between 1 and 100 in a very short amount of time.

Because computers are so well suited to repetitive tasks, programs are often written to repeat sequences of instructions.

An efficient way to repeat a series of instructions is to write a program loop. When a loop is executed, the series of statements in the loop is repeated until a terminating condition is met. While you can write your own program loops, the BASIC–PLUS language provides statements that are designed for building program loops. These statements are FOR, WHILE, UNTIL, and NEXT. The rest of this section shows you the advantages of program loops, describes their characteristics, and explains how to write them.

The best way to see the advantage of loops is by example. The following three program segments all perform the same function. Each prints a table of the square roots of the positive integers 1 through 100, together with their square roots. The first program segment does not use a loop, the second contains a user-written loop, and the third uses a built-in BASIC–PLUS feature called a FOR–NEXT loop.

Without a loop, the first program segment is 101 lines long and reads:

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

 .
 .
 .

990 PRINT 99, SQR(99)
1000 PRINT 100, SQR(100)
32767 END
```

The second example uses a user-written loop to obtain the same table:

```
10 LET X = 1
20 PRINT X,SQR(X)
30 LET X = X+1
40 IF X <=100 THEN 20
50 END
```

Statement 10 assigns a value of 1 to X, initializing the loop. In line 20, both the value of X and its square root are printed. In line 30, X is incremented by 1. Line 40 checks whether X is still less than or equal to 100; if so, the next value of X and its square root are printed.

You can also print the square root table with a FOR–NEXT loop:

```
10 FOR X = 1 TO 100
20 PRINT X, SQR(X)
30 NEXT X
40 END
```

Lines 10 through 30 of this program segment contain the FOR–NEXT loop. The FOR and NEXT statements in lines 10 and 30 cause the PRINT statement in line 20 to execute 100 times. After the number 100 and its square root are printed, X becomes 101. The condition in line 30 is now false, so control does not return to line 10. Control goes to line 40, which ends the program.

All program loops have four characteristic parts:

1. Initialization, the conditions that must exist for the first execution of the loop.
2. The body of the loop, where the operation to be repeated is performed.
3. Modification, which alters some value and makes each execution of the loop different from the one before and the one after.
4. Termination condition, an exit test that, when satisfied, completes the loop. Execution continues to the program statements after the loop.

BASIC–PLUS has two types of loops:

1. FOR–NEXT loops, which use a counter to determine how many times to repeat a series of instructions.
2. UNTIL–NEXT and WHILE–NEXT loops, which use a conditional statement to determine how many times to repeat a series of instructions.

As their names indicate, a FOR–NEXT loop contains a FOR statement, an UNTIL–NEXT loop contains an UNTIL statement, and a WHILE–NEXT loop contains a WHILE statement. All three loops contain a NEXT statement.

9.5.1 FOR and NEXT Statements

The FOR and NEXT statements together define a FOR–NEXT loop. The FOR statement begins the loop; the NEXT statement ends the loop. You code the body of the loop between them.

FOR–NEXT loops are controlled by a counter whose value is modified each time the body of the loop executes. This counter is called the *control variable* or *loop index*. You can let the control variable act only as a counter (and not reference it in body of the loop) or you can use the control variable inside the loop for processing.

The FOR statement has the form:

FOR <variable> = <expression> TO <expression> [STEP <expression>]

where:

<variable> is a numeric variable without a subscript

<expression> is a numeric expression

The variable in the FOR statement is the loop's control variable. While the control variable must be unsubscripted, it is common in the body of a loop to deal with subscripted variables, using the control variable as the subscript of a previously defined variable. (See Section 9.6 for information on subscripted variables.) For the greatest efficiency, use integer variables for control variables in the range –32767 to 32766. Use real variables for larger values or fractional values.

The expressions in the FOR statement must be numeric expressions as defined in Section 8.4. The first expression is the starting value for the control variable; the second is its final or terminal value. The optional STEP expression specifies the amount by which the control variable changes each time the loop is executed.

A positive STEP value increments the control variable; a negative STEP value decrements the control variable. If you omit the STEP expression, BASIC–PLUS assumes a value of +1. (Because +1 is a common STEP value, that portion of the statement is often omitted.)

An example of the FOR statement is:

```
10 FOR K% = 2% TO 20% STEP 2%
```

This statement cycles program execution through a loop using values for K% of 2, 4, 6, 8, and so on until 20. After the loop is executed with K% = 20%, the program exits from the loop. Control then passes to the line after the associated NEXT statement.

The NEXT statement signals the end of a loop that begins with a FOR statement. The NEXT statement has the form:

NEXT <variable>

The variable in a NEXT statement is the same one as in the FOR statement. Together the FOR and NEXT statements describe the boundaries of the program loop.

Upon encountering the NEXT statement, BASIC–PLUS adds the STEP value to the variable. Then it checks to see if the variable is still less than or equal to the terminal value. When the variable exceeds the terminal value, control falls through the loop to the statement following the NEXT statement.

The expressions within the FOR statement are evaluated before the initial entry into the loop. The test for completion of the loop is made before the loop is executed and at the end of each iteration of the loop. If the first test indicates completion, the loop is never executed.

Although you can modify the control variable within the loop, this practice is not recommended. When control falls through the loop, the control variable retains the last value used inside the loop. However, the terminal and STEP values are calculated only upon entry to the loop and do not change for the duration of the loop.

Consider the following program:

```
LISTNH
100 A=2.5
110 FOR I=1. TO 2.*A
120      PRINT I,A
130 A=1.
140 NEXT I

Ready

RUNNH

 1      2.5
 2        1
 3        1
 4        1
 5        1
```

The loop stops when I equals 5, even though the value of A changed within the loop.

The following examples show two loops that perform the same function. The loops are executed 10 times; the value of I is 10 when control leaves the loop (+1 is the assumed STEP value):

```
LISTNH
10 FOR I=1 TO 10
20 PRINT I;
30 NEXT I
40 PRINT I
```

```
LISTNH
10 I=1
20 PRINT I;
30 IF I<10 THEN &
      I=I+1 &
   \ GO TO 20
40 PRINT I
```

Both program segments produce the same result when run:

```
RUNNH
1  2  3  4  5  6  7  8  9  10  10
```

The numbers 1 through 10 are printed when the loop is executed. When I equals 10, control passes to line 40. Therefore 10, which is the current value of I, is printed again. Suppose line 10 had been:

```
10 FOR I = 10 TO 1 STEP -1
```

The value printed by line 40 would be 1. Note that a minus sign is necessary for decrementing a loop.

The following loop is executed only once, since the value of I = 44 is reached in the first iteration and the termination condition is satisfied.

```
10 FOR I = 2 TO 44 STEP 2
20 LET I = 44
30 NEXT I
```

If, however, the initial value of the variable is greater than the terminal value, the loop is never executed unless a negative STEP value is specified. A statement of the following format cannot be used to begin a loop:

```
10 FOR I = 20 TO 2 STEP 2
```

The statement that will execute the loop properly is:

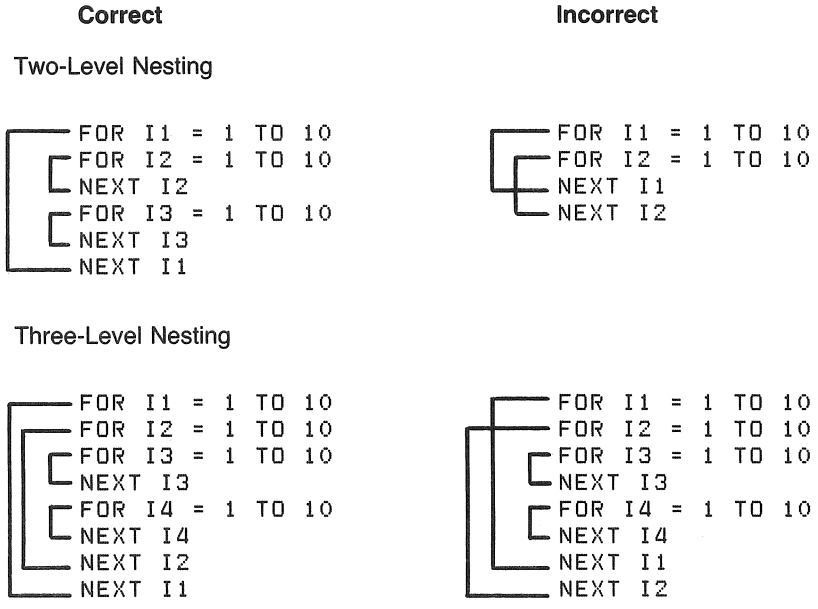
```
10 FOR I = 20 TO 2 STEP -2
```

For positive STEP values, the loop is executed until the control variable is incremented past its final value. For negative STEP values, the loop continues until decrementing the control variable causes the variable to be less than its final value.

FOR loops can be *nested* but not overlapped. Nesting is a programming technique where one or more loops are contained inside another loop. The range of one loop (the numbered lines from the FOR statement through the corresponding NEXT statement) must not cross the range of another loop.

The depth of nesting you can use depends on the size of the your program and the amount of memory that you have available. Figure 9–1 shows correct and incorrect nesting of loops.

Figure 9–1: Nesting Techniques



An example of nested FOR–NEXT loops follows:

```
LISTNH
100     FOR A%=1% TO 5%
110     FOR B%=2% TO 10% STEP 2%
120     PRINT A%;B%,
130     NEXT B%
140     PRINT
150     NEXT A%
32767   END

Ready

RUNNH
 1   2          1   4          1   6          1   8          1   10

 2   2          2   4          2   6          2   8          2   10

 3   2          3   4          3   6          3   8          3   10

 4   2          4   4          4   6          4   8          4   10

 5   2          5   4          5   6          5   8          5   10
```

You can exit from a FOR–NEXT loop without the control variable reaching the termination value by executing a GOTO, GOSUB, or computed GOTO. When you transfer control into a loop, you should only enter a loop left incomplete earlier. This transfer ensures that termination and STEP values are assigned. It is not good programming practice to enter loops anywhere but at the beginning of the loop.

Both FOR and NEXT statements can appear anywhere in a multi-statement line. For example:

```
LISTNH
100     FOR I=1 TO 10 STEP 5\ NEXT I \PRINT 'I=';I

Ready

RUNNH
I= 6

Ready
```

Neither the FOR nor NEXT statement is executed conditionally in an IF statement. The following statements are incorrect:

```
15 IF I<>J THEN NEXT I
15 IF I=J THEN FOR I=1 TO J
```

Both statements generate the error message:

```
?Illegal IF statement at line 15
```

NOTE

When the body of a loop contains only one statement, you can use FOR as a statement modifier to create a loop that is more efficient than a FOR–NEXT loop. See Section 13.6.3 for information about the FOR statement modifier.

9.5.2 WHILE and NEXT Statements

You can use the WHILE and NEXT statements to perform a loop while a specific condition continues to be true. A WHILE–NEXT loop is similar to a FOR–NEXT loop.

The WHILE statement has the form:

```
WHILE <condition>
```

where <condition> is a relational or logical expression.

The NEXT statement has the form:

```
NEXT
```

Do not include a variable in the NEXT statement.

The following is an example of a WHILE–NEXT loop. The sample program runs until you respond:

```
LISTNH
10 A$ = ""
20 WHILE LEN(A$) = 0%
30 INPUT "Enter name"; A$
40 A$ = CVT$$(A$,255%)
50 NEXT
32767 END

Ready

RUNNH
Enter name? RET
Enter name? RET
Enter name? RET
Enter name? JOE RET

Ready
```

Unlike a FOR–NEXT loop, a WHILE–NEXT loop does not have a control variable that is automatically incremented. However, a WHILE–NEXT loop works much like a FOR–NEXT loop. For example, the rules for termination are similar. Before the loop is executed, BASIC–PLUS tests to see if a condition is true. If it is, a pass through the loop is made. If the condition is false, the loop is not executed. Compare the following:

```
FOR I = 1 TO 5
NEXT I

WHILE I < 5
NEXT
```

The FOR and NEXT statements execute the loop five times; the WHILE and NEXT statements execute the loop until the value of I is no longer less than five.

Both FOR–NEXT and WHILE–NEXT loops follow the same rules for nesting.

NOTE

When the body of a loop contains only one statement, you can use WHILE as a statement modifier to create a loop that is more efficient than a WHILE–NEXT loop. See Section 13.6.4 for information about the WHILE statement modifier.

9.5.3 UNTIL and NEXT Statements

You can use the UNTIL and NEXT statements to perform a loop until a condition is true. An UNTIL–NEXT loop is similar to a FOR–NEXT loop and a WHILE–NEXT loop.

The UNTIL statement has the form:

UNTIL <condition>

where <condition> is a relational or logical expression.

The NEXT statement has the form:

NEXT

Do not include a variable in the NEXT statement.

The following example uses an UNTIL–NEXT loop. Like the example in the previous section, the sample program runs until you respond:

```
LISTNH
10 A$ = ""
20 UNTIL LEN(A$) > 0%
30 INPUT "Enter name"; A$
40 A$ = CVT$$(A$,255%)
50 NEXT
32767 END

Ready

RUNNH
Enter name? RET
Enter name? RET
Enter name? RET
Enter name? JIMRET

Ready
```

Like a WHILE–NEXT loop, an UNTIL–NEXT loop does not have a control variable that is automatically incremented. However, an UNTIL–NEXT loop works much like a FOR–NEXT loop. For example, the rules for termination are similar. Before the loop is executed, BASIC–PLUS tests to see if a condition is false. If it is, a pass through the loop is made. If the condition is true, the loop is not executed.

UNTIL–NEXT loops follow the same rules for nesting as FOR–NEXT and WHILE–NEXT loops.

NOTE

When the body of a loop contains only one statement, you can use UNTIL as a statement modifier to create a loop that is more efficient than an UNTIL–NEXT loop. See Section 13.6.5 for information about the UNTIL statement modifier.

9.6 Subscripted Variables and the DIM Statement

In addition to the simple variables described in Chapter 8, you can also use *subscripted variables* in BASIC–PLUS. A subscripted variable lets you break up a variable's storage area into several compartments. You can then refer to each compartment as a unique storage location. A set of compartments defined by a subscripted variable is called a *matrix* or, more commonly, an *array*.

You can use subscripted variables for additional computing capabilities with lists, tables, matrices, or any set of related variables. In BASIC–PLUS, subscripted variables can have one or two subscripts. BASIC–PLUS has default sizes for one- and two-dimensional arrays. However, you can specify the size of an array with the DIM statement.

The name of a subscripted variable is any acceptable BASIC–PLUS variable name, followed by one or two integer expressions in parentheses. If you use a floating-point number instead of an integer, the fractional portion is truncated. In other words, A(2.8) becomes A(2).

The following example lists A(I), where I goes from 0 to 5. (All arrays are created with a zero element, even if that element is never specified.)

```
A(0), A(1), A(2), A(3), A(4), A(5)
```

You can access each of six elements in the list, which can be shown as a one-dimensional array:

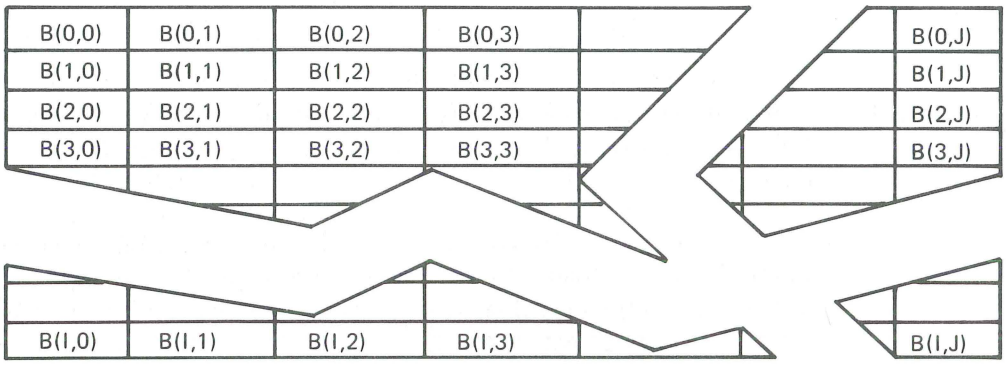
A(0)
A(1)
A(2)
A(3)
A(4)
A(5)

You can define two-dimensional array B(I,J) in a similar manner, as in Figure 9–2.

Subscripts used in subscripted variables can be constants or numeric expressions. You can use the same variable name as both a subscripted and an unsubscripted variable. Both A and A(I) are valid variables that can be

used in the same program without affecting one another. However, you cannot use the same variable name as both a singly and a doubly subscripted variable in the same program.

Figure 9–2: Array Structure



H-MK-00045-00

Consider the following example:

```
LISTNH
100 A=7
200 A(6)=14
300 PRINT A
400 PRINT A(6)

Ready

RUNNH
 7
 14

Ready
```

The preceding program segment is legal. If, however, you try to assign one variable name as both singly and doubly subscripted, the program does not run correctly:

```
100 A(6)=14
200 A(6,2)=7
```

Lines 100 and 200 generate the error message:

```
%Inconsistent subscript use at line 200
```

NOTE

There are cases where a variable name without subscripts refers to an entire array and not to a simple numeric variable. See the CHANGE statement, described in Section 10.2, and the MAT statements, described in Chapter 12.

The dimension (DIM) statement defines the maximum number of elements in an array. The DIM statement has the form:

DIM <variable(dimension(s))> [,<variable(dimension(s))> ,...]

The variables in a DIM statement are indicated with their maximum subscript value(s). For example:

```
10 DIM X(5), Y(4,2), A(10,10)
12 DIM I4(100)
```

You can use only nonnegative integer constants in DIM statements to define the size of an array. Any number of arrays can be defined in a single DIM statement as long as the variables defining them are separated by commas.

If you use a subscripted variable without using a DIM statement, BASIC–PLUS assumes the variable's highest element to be 10 in each dimension. The variable has elements 0 through 10 for a total of 11. However, you should correctly dimension all arrays in a program.

The first element of every array is automatically assumed to have a subscript of 0. Dimensioning A(6,10) sets up room for an array with 77 elements, arranged in 7 rows and 11 columns. The following program illustrates the 0 element:

```
LISTNH
10        REM - MATRIX CHECK PROGRAM
20        DIM A(6,10)
30        FOR I=0 TO 6
40        LET A(I,0)=I
50        FOR J=0 TO 10
60        LET A(0,J)=J
70        PRINT A(I,J);
          \ NEXT J
          \ PRINT
          \ NEXT I
32767     END

Ready

RUNNH
 0   1   2   3   4   5   6   7   8   9   10
 1   0   0   0   0   0   0   0   0   0   0
 2   0   0   0   0   0   0   0   0   0   0
 3   0   0   0   0   0   0   0   0   0   0
 4   0   0   0   0   0   0   0   0   0   0
 5   0   0   0   0   0   0   0   0   0   0
 6   0   0   0   0   0   0   0   0   0   0

Ready
```

Note that an array element, like a simple variable, has a value of 0 until it is assigned a value.

The size and number of arrays you can define depends on the amount of memory available in your program area. Additional information on arrays can be found in Chapter 12.

It is more efficient to dimension multiple arrays in one statement than to have multiple DIM statements.

You can place a DIM statement anywhere in a multi-statement line and anywhere in the program. It need not appear prior to the first reference to an array that it defines. DIM statements should be placed at or near the beginning of a program, however, to make them easy to change. You must not have two DIM statements referring to the same variable.

NOTE

For compatibility with BASIC–PLUS–2, it is recommended that you place the DIM statement before the first reference to the array it defines.

9.7 Mathematical Functions

Sometimes you need the computer to perform relatively common mathematical operations. You can often find the results of these operations (such as sine, cosine, square root, and log) in volumes of mathematical tables.

Several built-in mathematical functions are available as an optional BASIC–PLUS language feature. If your system has these built-in functions, you never have to consult tables or write your own code to find the value of the sine of 23 degrees or the natural log of 144. When you need these values in an expression, you can substitute the BASIC–PLUS functions. For example:

```
SIN(23.*PI/180.)
LOG(144.)
```

The various mathematical functions available in BASIC–PLUS are detailed in Table 9–1. These functions return a value in floating-point format. Note that these functions may not be available on all systems.

Most of these functions are self-explanatory; the rest are described in the following sections.

9.7.1 Sign Function, SGN(X)

The sign function returns a value of +1 if X is a positive value, 0 if X is 0, and –1 if X is negative. For example: SGN(3.42) = 1, SGN(–42) = –1, and SGN(23–23) = 0.

```
LISTNH
10       REM - SGN FUNCTION DEMO
100      READ A,B &
         \ PRINT 'A=';A, 'B=';B
110      PRINT 'SGN(A)=';SGN(A), 'SGN(B)=' SGN(B)
120      PRINT 'SGN(INT(A))=';SGN(INT(A))
1000     DATA    -7.32,  0.44
32767    END

Ready

RUNNH
A=-7.32          B= .44
SGN(A)=-1        SGN(B)= 1
SGN(INT(A))=-1

Ready
```

Table 9–1: Mathematical Functions

Function Code	Meaning
ABS(X)	Returns the absolute value of X.
SGN(X)	Returns the sign function of X, a value of 1 preceded by the sign of X, SGN(0) = 0.
INT(X)	Returns the largest integer less than or equal to X (INT(–.5) = –1) as a real number.
FIX(X)	Returns the value of X as an integer with any fractional portion removed (FIX(–.5) = 0).
COS(X)	Returns the cosine of X (X in radians).
SIN(X)	Returns the sine of X (X in radians).
TAN(X)	Returns the tangent of X (X in radians).
ATN(X)	Returns the arc tangent (in radians) of X.
SQR(X)	Returns the square root of X.
EXP(X)	Returns the value of e^X, where e = 2.71828...
LOG(X)	Returns the natural logarithm of X, $\log_e$ X.
LOG10(X)	Returns the common logarithm of X, $\log_{10}$ X.
PI	Has a constant value of 3.14159...
RND	Returns a random number between 0 and 1. Unless the RANDOMIZE statement appears in the program before the RND function, the program produces the same sequence of random numbers each time it runs. The value of X is ignored and can be omitted.

9.7.2 Integer Function, INT(X)

The integer function returns the value of the largest integer not greater than X. For example, INT(34.67) = 34. You can use INT to round numbers to the nearest integer with the expression INT(X + .5). For example, INT(34.67 + .5) = 35. You can also use INT to round to any given decimal place, for example:

```
INT(X*10.^ D+.5)/10.^D%
```

D is the number of decimal places desired. Consider the following program:

```
LISTNH
10      ! DEMONSTRATION OF INTEGER (INT) FUNCTION &
        ! INT DOES NOT ROUND TO NEAREST &
        ! INTEGER, BUT DROPS THE FRACTION PART
100     INPUT 'NUMBER TO BE PROCESSED BY INT FUNCTION';A
110     INPUT 'NUMBER OF DECIMAL PLACES FOR ROUNDING';D
120     PRINT 'TRUNCATED INTEGER=';INT(A)
130     PRINT 'ROUNDED INTEGER=';INT(A+.5)
140     PRINT 'ROUNDED TO ';D; 'PLACES='; &
                INT(A*10^D+.5)/(10^D)
150     PRINT
160     PRINT 'ENTER ANOTHER NUMBER, TYPE A ZERO TO STOP --'
170     INPUT A
180     IF A <> 0 GO TO 110
32767   END
Ready
```

```
RUNNH
NUMBER TO BE PROCESSED BY INT FUNCTION? 23.67
NUMBER OF DECIMAL PLACES FOR ROUNDING? 1
TRUNCATED INTEGER= 23
ROUNDED INTEGER= 24
ROUNDED TO 1 PLACES= 23.7

ENTER ANOTHER NUMBER, TYPE A ZERO TO STOP --
? 456.50505
NUMBER OF DECIMAL PLACES FOR ROUNDING? 2
TRUNCATED INTEGER= 456.
ROUNDED INTEGER= 457
ROUNDED TO 2 PLACES= 456.51

ENTER ANOTHER NUMBER, TYPE A ZERO TO STOP --
? 0

Ready
```

For negative numbers, the largest integer contained in the number is a negative number with the same or a larger absolute value. For example: INT(–23) = –23, but INT(–14.39) = –15.

9.7.3 Random Number Function, RND

The random number (RND) function produces a random number greater than or equal to 0 but less than 1. You can reproduce the numbers in the same order for later checking of a program. You do not need an argument with the RND function, although you may use one. This argument can be any number, since its value is ignored. For compatibility with BASIC–PLUS–2, you should not use an argument with the RND function.

```
LISTNH
10      REM - RANDOM NUMBER DEMONSTRATION
20      INPUT "HOW MANY RANDOM NUMBERS";N
30      FOR I=1 TO N
40      PRINT RND,
50      NEXT I
32767   END

Ready

RUNNH
HOW MANY RANDOM NUMBERS? 13
 .204935        .229581         .533074         .132211         .995602
 .783713        .741854         .397713         .709588         .67811
 .682372        .991239         .806084
Ready
```

To obtain random digits from 0 to 9, change line 40 to read:

```
40      PRINT INT(10*RND),
```

Then run the program again. The results of the second run are:

```
RUNNH
HOW MANY RANDOM NUMBERS? 13
 2              2               5               1               9
 7              7               3               7               6
 6              9               8
Ready
```

You can generate random numbers over any range. In general, if you want the range (A,B), use either:

```
(B-A)*RND(X)+A

(B-A)*RND+A
```

This produces a random number in the range A<n<B.

9.7.4 RANDOMIZE Statement

The RANDOMIZE statement has the form:

RANDOM[IZE]

Use the RANDOMIZE statement with the RND function to obtain different random numbers every time you run a program. Place RANDOMIZE before the first RND function in the program. When executed, RANDOMIZE causes the RND function to choose a random starting value, so that the same program run twice gives different results. For this reason, it is a good practice to debug a program completely before inserting the RANDOMIZE statement.

To demonstrate the effect of the RANDOMIZE statement on two runs of the same program, the RANDOMIZE statement appears in statement 15 of the following program:

```
LISTNH
10      REM - RANDOM NUMBER DEMONSTRATION
15      RANDOMIZE
20      INPUT "HOW MANY RANDOM NUMBERS";N
30      FOR I=1 TO N
40      PRINT RND,
50      NEXT I
32767   END

Ready

RUNNH
HOW MANY RANDOM NUMBERS? 35
 .541559        .249281        .621653        .486387        .32345
 .563214        .468236        .740494        .228836        .708573
 .191913        .774316        .918683        .543249        .99135
 .058857        .430996        .562636E-1     .458616        .245325
 .344403        .858497        .513523E-1     .581639        .276619E-1
 .931222        .338373        .649245        .850108        .257448
 .893713        .452437E-1     .228047        .961087        .714104

Ready

RUNNH
HOW MANY RANDOM NUMBERS? 12
 .448782        .692627        .116732        .466741        .749863
 .29851         .422888E-1     .567144        .022262        .292799E-1
 .975321        .588409

Ready
```

The output from each run is different.

9.8 User-Defined Functions

Sometimes you want a program to execute the same sequence of statements or mathematical formulas in several different places. BASIC–PLUS allows you to define your own functions and call them in the same way you call the standard functions, such as RND, SQR, or COS.

A user-defined function name consists of FN followed by any valid variable name. For example:

```
FNA
FNA1
```

Because A and A1 are real variables, FNA and FNA1 are real functions. They return floating-point values. You can also define string functions (which return string values) and integer functions (which return integer values). To do so, append a string or integer variable name to FN. For example:

```
FNA$
FNA%
```

Like variables, functions with the same name but different suffixes are different functions. See Chapters 10 and 11 for more information on string and integer functions.

You can define a function anywhere in the program. Write the defining, or DEF*, statement as:

DEF* FN<variable(arguments)> = <expression(arguments)>

NOTE

For compatibility with previous versions of BASIC–PLUS, DEF is supported as a synonym of DEF*. However, DEF* is preferred for compatibility with BASIC–PLUS–2.

The arguments in a DEF* statement can consist of zero to five dummy variables. The expression, however, need not contain all the arguments and can also contain other program variables not among the arguments. For example:

```
10 DEF* FNA(S) = S^2
```

The preceding statement causes line 20 to be evaluated as R = 17:

```
20 LET R = FNA(4)+1
```

The following example causes the function to be evaluated using the current value of the variable X in the program:

```
50 DEF* FNB(A,B) = A+X^2
60 Y=FNB(14,4,R3)
```

The dummy argument in this case (B, which becomes the actual argument R3 in the function call) is not used.

The following two programs illustrate the use of a user-defined function. The first program uses different variables for the dummy argument in the DEF* statement and the actual argument in the function call; the second program uses the same variable for both the dummy argument and the actual argument. Otherwise, the two programs are identical. Both produce the same output.

Program 1

```
LISTNH
10     ! DEMO OF FUNCTION DEFINITION
100    DEF* FNS(A)=A^A
110    FOR I=1 TO 5 &
       \ PRINT I, FNS(I) &
       \ NEXT I
32767 END

Ready
```

Program 2

```
LISTNH
10     ! DEMO OF FUNCTION DEFINITION
100    DEF* FNS(I)=I^I
110    FOR I=1 TO 5 &
       \ PRINT I, FNS(I) &
       \ NEXT I
32767 END

Ready
```

The output is:

```
RUNNH
 1             1
 2             4
 3             27
 4             256
 5             3125

Ready
```

DEF* statement arguments are dummy variables. When your program invokes the defined function, the first value passed in the function call is assigned to the first dummy variable. The second value is passed to the second dummy variable, and so forth. The number of values passed must match the number of dummy variables in the DEF* statement:

```
100    DEF* FNA(X) = X^2+3*X+4
200    DEF* FNB(X) = FNA(X)/2 + FNA(X)
300    DEF* FNC(X) = SQR(X+4) + 1
```

The arguments of a user-defined function can be numbers, variables, other functions, or mathematical expressions. For example:

```
40 LET R = FNA(X+Y+Z)*N/(Y^2+D)
```

A user-defined function can have zero to five arguments:

```
25 DEF* FNL(X,Y,Z) = SQR(X^2 + Y^2 + Z^2)
```

A later statement in a program containing this user-defined function might look like:

```
55 LET B = FNL(D,L,R)
```

where D, L, and R have some values in the program.

The number of arguments you use to call a user-defined function must agree with the number of arguments that defined it. Otherwise, you get an error message. For example:

```
10 DEF* FNA (X) = X*2
20 PRINT FNA(3,2)
```

These statements cause the error message:

```
?Arguments don't match at line 20
```

You can use a DEF* statement or function reference, where a function has zero arguments, to write the function name with or without parentheses. For example:

```
LISTNH
10      DEF* FNA=X^2
20      INPUT 'TYPE A NUMBER';X
30      PRINT FNA; FNA()
32767   END

Ready

RUNNH
TYPE A NUMBER? 3.65
 13.3225   13.3225

Ready
```

When you call a user-defined function, you can use any legal expression as an argument. BASIC–PLUS substitutes the value of each expression for the corresponding function variable. For example:

```
LISTNH
10      DEF* FNZ(X)=X^2
20      LET A=2 &
        \PRINT FNZ(2+A)
32767   END

Ready

RUNNH
 16

Ready
```

If you define the same function name more than once, an error message is printed:

```
100     DEF* FNX(X)=X^2
110     DEF* FNX(X)=X^4
?Illegal FN redefinition at line 110

Ready
```

The function variable need not appear in the function expression. For example:

```
LISTNH
100     ! FUNCTION VARIABLE NOT IN FUNCTION EXPRESSION
110     DEF* FNA(X) = 4*A + 2
120     FOR A = 0 TO 3
130     LET R = FNA(10) + 1 &
        \ PRINT R &
        \ NEXT A
32767   END

Ready

RUNNH
 3
 7
 11
 15

Ready
```

The next program contains examples of multi-variable DEF* statements in lines 30, 50, and 70.

```
LISTNH
30      DEF* FNA(X,Y) = X + Y
50      DEF* FNM(X,Y) = X * Y
70      DEF* FNU(A,B,C,D,E)=((A-B)*C)+(D-A)/(E+4)
100     INPUT "2 numbers to ADD"; A,B
110     PRINT "The sum of"; A; "+"; B; "is"; FNA(A,B)
200     INPUT "2 numbers to MULTIPLY"; E,F
210     PRINT "The product of"; E; "*"; F; "is"; FNM(E,F)
300     INPUT "5 numbers to play with"; A,B,C,D,E
310     PRINT "The unusual result of that is"; FNU(A,B,C,D,E)
32767   END

Ready
```

```
RUNNH
2 numbers to ADD? 5,6
The sum of 5 + 6 is 11
2 numbers to MULTIPLY? 4,5
The product of 4 * 5 is 20
5 numbers to play with? 6,2,3,14,8
The unusual result of that is 12.6667

Ready
```

9.9 Subroutines

A subroutine is a section of code that can be called at more than one point in the program. A subroutine can perform a complicated I/O operation for a volume of data, a mathematical evaluation, or any number of other operations. The first line in the subroutine can be a remark or any executable statement; the remaining lines in the subroutine perform whatever operation you specify.

Like user-defined functions, subroutines let you execute the same sequence of instructions at more than one point in a program. Unlike user-defined functions, which are called by name, subroutines are called by line number.

You can use more than one subroutine in a single program. They are normally placed in line number sequence at the end of the program.

A useful practice is to assign distinctive line numbers to subroutines. For example, if the main program uses line numbers up to 199, use 200 and 300 as the first numbers of two subroutines. Consider the following subroutine that uses both the GOSUB and RETURN statements:

```
LISTNH
10      PRINT "HI THERE." &
        \ GOSUB 1000 &
        \ PRINT "I'M BACK AT LINE 10" &
        \ GOSUB 1000

20      PRINT "I'M BACK AT LINE 20 NOW" &
        \ GOTO 32767

1000    PRINT "JUMPING JEHOSOPHAT!" &
        \ RETURN

32767   END

Ready

RUNNH
HI THERE.
JUMPING JEHOSOPHAT!
I'M BACK AT LINE 10
JUMPING JEHOSOPHAT!
I'M BACK AT LINE 20 NOW

Ready
```

9.9.1 GOSUB Statement

Subroutines usually are placed near the end of a program before any DATA statements and must go before the END statement. The GOSUB statement calls the subroutine. Your program continues until it encounters a GOSUB statement of the form:

GOSUB <line number>

The line number after the word GOSUB is the first line number of the subroutine. Control then transfers to that line in the subroutine. For example:

```
50 GOSUB 200
```

Control transfers to line 200 in your program subroutine.

9.9.2 RETURN Statement

A RETURN statement is necessary to exit a subroutine. It has the form:

RETURN

You use RETURN in the following sequence:

1. The program encounters a GOSUB statement.
2. The system internally records the location of the next executable statement.
3. Control transfers to the subroutine.
4. A RETURN statement at the end of the subroutine transfers control back to where it left off in the main program.

In this way, no matter how many times you call subroutines, the program returns to the right place. For example, refer to the subroutine in Section 9.9.

9.9.3 Nesting Subroutines

Subroutines can be nested; that is, one subroutine can call another subroutine. If the execution of a subroutine encounters a RETURN statement, it returns control to the statement after the GOSUB that called the subroutine. Therefore, a subroutine can call another subroutine, including itself.

Subroutines can be entered at any point and can have more than one RETURN statement. You can transfer to the beginning or to any part of a subroutine. Multiple entry points and returns can make a subroutine more versatile, although more difficult to debug and maintain.

The maximum level of GOSUB nesting depends on the size of your program and the amount of memory available. Exceeding this limit generates a message of the form:

```
?Maximum memory exceeded at line <line number>
```

9.10 END Statement

The END statement is the last statement in a BASIC–PLUS program. When executed, it closes all open I/O channels and halts program execution.

While BASIC–PLUS lets you omit the END statement, you should always include it to guarantee an orderly end to program execution.

The END statement has the form:

```
END
```

Its line number should be the largest line number in the program. Programmers often place the END statement at line 32767 (the largest line number allowed) to guarantee that it is the last line in the program.

BASIC–PLUS lets you create a program that has statements after the END statement. However, when you run or retrieve this program, BASIC–PLUS ignores all statements that follow the END statement. Suppose, for example, that you write a program where an END statement is followed by additional statements:

```
NEW TEST

Ready

100 PRINT "HELLO"
200 PRINT "GOODBYE"
300 END
400 LET C = 5
500 PRINT C
```

You see all the lines when you list the program, but lines 400 and 500 do not execute when you run the program. Similarly, you can save the program, but lines 400 and 500 are lost when you use the OLD command to retrieve the program. For example:

```
SAVE TEST

Ready

OLD TEST

Ready

LISTNH
100 PRINT "HELLO"
200 PRINT "GOODBYE"
300 END

Ready
```

You can run a program that does not contain an END statement. BASIC–PLUS executes an implicit END statement and does not print an error message. However, BASIC–PLUS prints the following message when you retrieve a program without an END statement:

```
?End of file on device
```

9.11 STOP Statement

Like the END statement, the STOP statement halts program execution. Unlike the END statement, however, the STOP statement does not close I/O channels. In addition, the STOP statement does not usually appear in a finished BASIC–PLUS program. Instead, is a debugging tool. STOP lets you halt execution at various points in a program to examine variables and make changes to the program.

The STOP statement has the form:

STOP

When executed, STOP halts the program and prints:

```
Stop at line <line number>

Ready
```

The <line number> is the line containing the STOP statement. Chapter 6 of this manual describes the use of the STOP statement in program debugging.

Usage Note

When you debug an error-handling routine, be aware that STOP resets the values of the BASIC–PLUS variables ERR and ERL, which are used in error-handling routines. STOP sets ERR (the error number) to 123, which is the "Stop at line <line number> " message. STOP sets ERL, (the line where the error occurred), to the line containing the STOP statement.

See Section 13.7 for information about error-handling routines.

PART III
Advanced Language Features

Chapter 10
Strings and String Functions

This chapter reviews string constants and variables and describes string conversion, string input and output, and string functions.

10.1 Strings

Chapter 9 describes the manipulation of numeric information. BASIC–PLUS also processes information in *strings*. A string is a sequence of ASCII characters treated as a unit. Notice the messages that the INPUT and PRINT statements print in Sections 9.3.2 and 9.3.3 with the input and output of numeric values. These messages consist of string constants. Just as there are numeric constants and functions, there are also string constants and functions.

10.1.1 String Constants

BASIC–PLUS lets you use string constants as well as numeric constants. String constants are delimited by either single or double quotation marks. For example:

```
100    LET Y$ = "FILE4"
200    B1$ = 'CAN'
300    IF A$ = "YES" GOTO 250
```

"FILE4", 'CAN' and "YES" are string constants.

10.1.2 String Variables

You can define variable names for simple strings and also for lists and matrices composed of strings (one- and two-dimensional string arrays). Any legal variable name followed by a dollar sign ($) is a legal name for a string

variable. (Section 8.3.2 describes the rules for forming legal variable names.) For example:

```
A$
C7$
NAME.OF.CUSTOMER$     (EXTEND mode only)
```

These are simple string variables. Any subscripted variable name followed by a dollar sign ($) denotes a string array. For example:

```
V$(N)     M2$(N)
C$(M,N)   G1$(M,N)
```

M and N indicate the position of the array element in the array.

BASIC–PLUS automatically initializes numeric variables to zero when you run a program. Likewise, it initializes string variables to null strings (strings containing no characters).

10.1.3 Subscripted String Variables

Define string arrays with the DIM statement, as you do for numeric arrays. For example:

```
100    DIM S1$(5)
```

This statement means that S1$ is a string array with six elements, S1$(0) through S1$(5), which you access individually. If you do not use a DIM statement, BASIC–PLUS assumes that a subscripted string variable has a dimension of 10 in each direction (11 elements including zero). Note that the dimension of a string array specifies the number of strings, not the number of characters in any string. For example, consider that the first statements in a program are:

```
1050    FOR I=1 TO 7 &
        \ LET B$(I)="PDP-11" &
        \ NEXT I
```

BASIC–PLUS creates an array B$(n) with 11 accessible elements, B$(0) through B$(10). The elements B$(1) through B$(7) are set equal to "PDP–11". The others are null strings:

```
LISTNH
1050    FOR I=1 TO 7 &
        \ LET B$(I)="PDP-11" &
        \ NEXT I
1060    FOR H=0 TO 10 &
        \ PRINT H,B$(H) &
        \ NEXT H
32767   END

Ready
```

```
RUNNH
 0
 1        PDP-11
 2        PDP-11
 3        PDP-11
 4        PDP-11
 5        PDP-11
 6        PDP-11
 7        PDP-11
 8
 9
 10

Ready
```

In general, dimension all arrays to the maximum size that the program refers to.

10.1.4 String Size

A string can contain up to 32767 characters, but is limited by the amount of available memory.

You cannot use the LINE FEED key to type a string on two or more terminal lines. To create a string longer than a terminal line, use string operations (such as concatenation and other operations that are described later). Since memory is limited, you can also save strings in disk files.

10.1.5 Relational Operators

When you apply relational operators to string operands, the operators indicate alphabetical order. For example:

```
55    IF A$(I)<A$(I+1) GOTO 100
```

A$(I) and A$(I+1) are compared at line 55. If A$(I) occurs earlier in alphabetical order than A$(I+1), execution continues at line 100. See Section 8.4.3.2 for more information on string relational expressions.

10.2 ASCII String Conversions, CHANGE Statement

You can reference individual characters in a string with the CHANGE statement. It has the form:

CHANGE { <array name> / <string variable> } TO { <string variable> / <array name> }

The CHANGE statement lets you convert either a string into a list of numeric values or a list of numeric values into a string. You can convert each character in a string to its ASCII equivalent or vice versa.

Table D–1 lists the ASCII characters and their corresponding decimal values. Several ASCII characters have no graphic equivalent; they do not cause a character to be printed.

The following program uses CHANGE to convert a string into a list of numeric values. The program stores the numeric values in an integer array.

```
LISTNH
1000    REM -- STRING/ASCII CHANGE DEMO
1010    DIM X%(3)
1020    LET A$ = "CAT"
1030    CHANGE A$ TO X%
1040    X%=X%(0)+X%(3) &
        \ PRINT 'X%=';X%,'THE ARRAY X% IS '; &
                X%(0);X%(1);X%(2);X%(3)
1050    !IN A CHANGE STATEMENT THE NUMERIC ARRAY &
        !IS REFERENCED WITHOUT SUBSCRIPTS, AS THIS &
        !EXAMPLE SHOWS, HAVING A SINGLE-VALUE VARIABLE &
        !WITH THE SAME NAME IN THE PROGRAM CAUSES &
        !NO AMBIGUITY.
32767   END

Ready

RUNNH
X%= 87       THE ARRAY X% IS 3  67  65  84
```

X%(1) through X%(3) contain the ASCII values of the characters in the string variable A$. The first element of array X%, X%(0), becomes the number of characters in A$.

If more characters are present in the string variable than are dimensioned in the numeric list, the message "?Subscript out of range" is printed. The first element of the list (element 0) becomes the number of characters in the string and is greater than the dimension of the list.

Notice that line 1010 creates the four-element array, X%. Use a DIM statement in this instance. Otherwise, BASIC–PLUS creates a default eleven-element array (X%(10)), which wastes space in your memory area.

Another program that converts a string into a list of numeric values follows:

```
LISTNH
100     DIM A%(65)
110     READ B$
120     CHANGE B$ TO A%
130     FOR I=0 TO A%(0) &
        \ !A%(0) GIVE LENGTH OF STRING
140     PRINT A%(I); &
        \ NEXT I
200     DATA ABCDEFGHIJKLMNOPQRSTUVWXYZ
32767   END

Ready
```

```
RUNNH
 26  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81
 82  83  84  85  86  87  88  89  90
Ready
```

Notice that A%(0) equals 26, the number of letters in the alphabet.

So far, you have seen how to convert a string into a list of numeric values. You can also use the CHANGE statement to convert the numeric (ASCII) representation of characters into string characters.

The following example uses CHANGE to convert an array of ASCII values into a string. A%(0) specifies the number of characters in the string. The ASCII values for the characters start at A%(1).

```
LISTNH
100    REM -- ASCII NUMBER TO STRING CONVERSION DEMO
110    FOR I=0 TO 5 &
       \ READ A%(I) &
       \ NEXT I
120    CHANGE A% TO B$ &
       \ PRINT B$
200    DATA 6,65,66,67,68,69,70
32767  END

Ready

RUNNH
ABCDE

Ready
```

This program prints ABCDE because the numbers 65 through 69 are the ASCII code numbers for A through E. Although the DATA statement contains a value for A%(6), the program does not read it into the array. Thus, only five characters (ABCDE) are printed. ASCII 0 (the contents of A%(6)) is not a printable character.

When you use CHANGE to convert an array to a string, you must indicate the number of characters in the string. Place this number in the zero element of the array. If element zero has a value less than or equal to zero, the CHANGE statement generates a zero-length string.

NOTE

The CHANGE statement is different from the intrinsic functions VAL(X$) (which converts a number's string representation to a number) and NUM$(X) (which converts a number to its string representation). See Table 10–1 for information about the VAL and NUM$ functions.

10.3 String Input

You can input strings with READ and DATA, INPUT, and INPUT LINE statements.

10.3.1 READ and DATA Statements

The READ and DATA statements enter string variables into a program. A READ statement can appear anywhere in a multi-statement line, but a DATA statement must be the last statement on a line.

Here is an example of READ and DATA:

```
10 READ A$, B, C, D
20 DATA 17, 14, 13.4, CAT
```

These statements cause BASIC–PLUS to make the following assignments:

A$ = the character string "17"
B = 14
C = 13.4

Attempting to read CAT into D causes BASIC–PLUS to print the message:

```
?Illegal number at line 10
```

You need quotation marks (") around a string item in a DATA statement only when:

- The string contains a comma
- Leading, trailing, or embedded spaces within the string are significant
- Lowercase letters should not be converted to uppercase

BASIC–PLUS always accepts single or double quotation marks around string items, even when they are not necessary. For example, the items in line 500 in the following program are acceptable strings:

```
LISTNH
100    READ A$,B$,C$,D$,E$
110    PRINT A$;B$;C$;D$;E$
120    PRINT A$,B$,C$,D$,E$
500    DATA 'MR. JONES', MS SMITH, "MRS,BROWN", 'MS', '"MR"'
32767 END

Ready

RUNNH
MR. JONESMSSMITHMRS,BROWNMS"MR"
MR. JONES    MSSMITH   MRS,BROWN MS      "MR"

Ready
```

Although the string MS SMITH is acceptable without quotation marks, embedded spaces in the string are discarded.

BASIC–PLUS stores all values from the programmed DATA statements as an ASCII string list. When it reads a numeric variable, BASIC–PLUS performs the appropriate ASCII–to–numeric conversions. When a string variable is read, the string is used as it appears in the DATA statement. If the item is not in quotation marks, BASIC–PLUS ignores leading, trailing, and embedded spaces. If the item appears in quotation marks, the string variable is equated to the string within the quotation marks.

See Section 9.2.3 for more information about READ and DATA. See also the MAT READ statement, Section 12.2. MAT READ reads numeric and string matrices.

10.3.2 INPUT Statement

You can use the INPUT statement to input strings. For example:

```
10 INPUT "YOUR NAME";N$
```

This statement is equivalent to:

```
100    PRINT 'YOUR NAME';
110    INPUT N$
```

When an INPUT statement is executed, it prompts you for input with a question mark. To ensure that BASIC–PLUS reads your input correctly, it is recommended that you enter no more than 132 characters (including commas between values) before you press the RETURN key. See Sections 9.2.2 and 15.4 for more information about the INPUT statement.

10.3.3 INPUT LINE Statement

Besides INPUT, you can also input strings with the statement:

INPUT LINE <string variable>

For example:

```
10 INPUT LINE A$
```

This statement causes the program to accept a line of input from the terminal with embedded spaces, punctuation characters, or quotation marks.

You cannot print a text string with the INPUT LINE statement. For example:

```
10    INPUT LINE 'YOUR NAME'; A$
?Syntax error at line 10
```

To print a text string, use the PRINT statement.

An INPUT LINE statement reads the line as you type it, including the line terminating character(s). You can terminate the line in one of the following ways:

1. *Carriage return/line feed*—press the RETURN key (appends ASCII values 13 and 10 to the character string).

2. *Line feed*—press the LINE FEED key (appends ASCII values 10, 13, and 0 to the character string).

3. *ESCAPE*—press the ESCAPE, ALT MODE or PREFIX key, depending on the terminal (appends ASCII value 27 to the character string).

4. *Form feed*—press CTRL/L (appends ASCII value 12 to the character string).

When BASIC–PLUS encounters an INPUT or INPUT LINE statement, it displays a prompt (a question mark and a space). This signals that the next line typed will be treated as input.

You can enter up to 132 characters. If you enter more than 132 characters, BASIC–PLUS may not read them correctly. Your string variable will have unpredictable contents. For example:

```
LISTNH
100     PRINT 'ENTER LONG STRING' &
        \ INPUT LINE A$ &
        \ PRINT &
        \ PRINT A$
32767   END

Ready

RUNNH
ENTER LONG STRING
? ,./[';;']":]\PO[\OIJKMNH FFFFFFFFFFFFFFF5555555556666677778889011111122
233344RRFFCDSSWWEERGNNM,,KKLL:::::DDDD#########________++++++UUUIKOLBBBB
BBBFFFFFFFFFFFFFFRRRRRRRRR44444445555556? 66666777777888888999IIIIIIIUUU
UYYYYFFFFFFVVVVVBBBBNNNNMMMVV? VVCCCSSSSAAAAQQQSSSSWWWEEDDD4444444444444
4444%%%%%%%%^^^^^^&&&&&&$$$$$$#####@@@@!!!!!HHHHH6666667777778888?Line t
oo long at line 100
? HH

4444444444444444%%%%%%%%^^^^^^&&&&&&$$$$$$#####@@@@!!!!!HHHHHH

Ready
```

You can also input string information in the form of records with the GET statement. See Section 17.3 for further information.

10.4 String Output

Use the PRINT statement to print string data. When a PRINT statement includes string values (either alone or with other string and numeric values), BASIC–PLUS does not add leading or trailing spaces to the string. Only the actual content of the string is printed. If you want to print leading or trailing spaces, you must include them in the string when you define it. The following example shows how BASIC–PLUS prints strings:

```
LISTNH
100    REM -- STRING OUTPUT DEMO
200    A$="PART 1"
300    B$="PART 2"
400    C$="PART 3"
500    PRINT A$;B$;C$;
32767  END

Ready

RUNNH
PART 1PART 2PART 3
Ready
```

The use of semicolons to separate character string constants from other list items is optional in BASIC–PLUS. However, their use is recommended because they make the program more transportable and readable.

10.5 String Functions

BASIC–PLUS provides built-in mathematical functions for numeric quantities (for example, SIN and LOG). Similarly, the language offers a set of functions to simplify the handling of strings. These functions, described in Table 10–1, are particularly useful when dealing with whole lines of alphanumeric information input by an INPUT LINE statement. Throughout Table 10–1, A$ in the immediate mode examples is the string: "ABCDEFGHIJKLMNOPQRSTUVWXYZ".

Table 10–1: String Functions

Function Code	Meaning

LEFT(A$,N)	Indicates a substring of the string A$ from the first character through the Nth character (the leftmost N characters of the string A$). For example:
	`A$='ABCDEFGHIJKLMNOPQRSTUVWXYZ'`
	`Ready`
	`PRINT LEFT(A$,7)`
	`ABCDEFG`

(continued on next page)

Table 10–1: String Functions (Cont.)

Function Code	Meaning

RIGHT(A$,N)	Indicates a substring of the string A$ from the Nth character through the last character in A$ (the rightmost characters of the string A$ starting with the Nth character). For example:
	
	`PRINT RIGHT (A$,20%)`
	`TUVWXYZ`

MID(A$,N1,N2)	Indicates a substring of the string A$, starting with character N1, that is N2 characters long (the characters between and including the N1 through N1+N2–1 characters of the string A$). For example:
	
	`PRINT MID (A$,15%,5%)`
	`OPQRS`

LEN(A$)	Returns an integer that indicates the number of characters in the string A$ (including trailing blanks). For example:
	
	`PRINT LEN (A$)`
	`26`

+	Indicates a concatenation operation on two strings. For example, "ABC"+"DEF" is equivalent to "ABCDEF", and "12"+"34"+"56" is equivalent to "123456".
CHR$(N)	Generates a one-character string with the ASCII value of N. For example, CHR$(65) is equivalent to "A". Only one character can be generated.
ASCII(A$)	Generates an integer that is the ASCII decimal value of the first character in A$. For example, ASCII("X") is equivalent to 88, the ASCII equivalent of X. If B$ = "XAB", then ASCII(B$) = 88.
RAD$(I%)	Converts an integer to a three-character string. This function lets you convert a value in Radix–50 format back into ASCII. (Radix–50 is explained in the *RSTS/E Programming Manual*.)

INSTR(N1,A$,B$)	Indicates a search for the substring B$ within the string A$, beginning at character position N1. Returns a value of 0 if B$ is not in A$. Returns the character position of B$ if B$ is found in A$. (Character position is measured from the start of the string, with the first character counted as character 1.) For example:
	
	`PRINT INSTR(5%,A$,'OP')`
	`15`
	
	If B$ (the search string) is a null string (B$ = ""), the INSTR function returns the value 1. The null string is a proper substring of any string; it is treated conventionally as the first element of A$ in null string search operations. In addition, if both A$ and B$ are null strings, the INSTR function returns the value 1.

SPACE$(N)	Indicates a string of N spaces; used to insert spaces in a character string.

(continued on next page)

Table 10–1: String Functions (Cont.)

Function Code	Meaning

NUM$(N)	Returns a string of numeric characters representing the value of N as a PRINT statement would print it. NUM$(n) = (space)n(space) if n>0 and NUM$(n) = –n(space) if n<0. For example:
	`PRINT NUM$(7465098702134)`
	`.74651E 13`

NUM1$(N)	Returns a string of numeric characters representing the integer or floating-point value N. This is similar to the NUM$ function, except that no spaces or E-format results are returned. You can use it to convert an integer or floating-point value for use as a string function operand. For example:
	`PRINT NUM1$(PI)`
	`3.14159265358979`
	This example uses a four-word format, which allows you to carry out PI to 15 significant digits. Using a two-word format, PI would equal 3.14159.
	Here is another example:
	`PRINT NUM1$(97.5*30456.23+3.03^5.1)`
	`296976.76154965`

VAL(A$)	Computes the numeric value of the string of numeric characters A$ and returns it as a floating-point number. A$ may include digits, a leading plus (+) or minus (–), a period (.) and E. For example:
	`PRINT VAL('14.3E-5')`
	`.000143`
	If A$ contains a character not acceptable as numeric input with the INPUT statement, an error results.

TIME$(N)	Where N = 0, this function returns the time of day as a string. For example: `01:30 PM` Where N<>0, the function translates N% into a time string. If the run-time system was generated using the 24–hour time option, 01:30 PM is returned as 13:30 followed by 3 spaces. TIME$ always returns an 8–character string.
DATE$(N)	Where N = 0, this function returns the current date in the form: <day>–<month>–<year>. For example: `12-Aug-81` Note that dates are output using both upper- and lowercase letters. When the output device does not generate lowercase letters, the ASCII values still imply lowercase. Where N<>0, the function translates N into a date string. If the run-time system was generated with the numeric date option, 12–Aug–81 is returned as 81.08.12.

(continued on next page)

Table 10–1: String Functions (Cont.)

Function Code	Meaning

STRING$(N1,N2)	Creates a string of length N1 that consists of characters whose ASCII decimal value is N2. For example, to create a string composed of 10 "A" characters (CHR$(65)), execute the statement:
	`PRINT STRING$(10,65)`
	`AAAAAAAAAA`
	See Table D–1 for the decimal values of ASCII characters.

CVT$$(S$,M)	Converts the source character string S$ according to the decimal value of the integer M. Section 10.5.1 describes this function in detail.
XLATE(S$,T$)	Translates source string S$ from its existing storage code to a code indicated by the table string T$, and returns the translated form of string S$ as the target string. A complete description of this function appears in Section 10.5.2.

10.5.1 CVT$$ Function

The CVT$$ function manipulates a character string and generates a new character string. The output string is converted according to the integer value that your program gives. You specify CVT$$ in the form:

CVT$$ (S$,M%)

where:

S$ is the string to convert

M% is an integer value

The bits of M% are interpreted as follows:

1 Trim the parity bit.

2 Discard spaces and tabs.

4 Discard carriage return (RET), line feed (LF), form feed (FF), escape (ESC), rubout (DEL), and fill or null characters.

8 Discard leading spaces and tabs.

16 Reduce spaces and tabs to one space.

32 Convert lowercase to uppercase.

64 Convert square brackets to parentheses; for example, "[" to "(" and "]" to ")".

128 Discard trailing spaces and tabs.

256 Do not allow alteration of characters inside single or double quotation marks except parity bit trimming.

You can use these bits in combination. For instance, if M% is given as 21%, the result is the same as if you had used three CVT$$ functions with M% values of 1%, 4%, and 16%.

The value 1% in the CVT$$ function removes the parity bit (most significant bit) from each character in the string. Under RSTS/E, characters are usually represented with no parity. All comparison of characters assume no parity.

The value 2% removes all space characters (CHR$(32)) and horizontal tab characters (CHR$(9)) from the string. Values 8%, 16%, and 128% remove only selective occurrences of space and horizontal tab characters. The terminating and excess characters which the value 4% removes in the CVT$$ function usually have no informational value in a string.

For example:

```
A$="    abcde     "
    TAB       5 spaces

Ready

PRINT CVT$$(A$,2%)
abcde

Ready

PRINT CVT$$(A$,16%)
 abcde
```

The value 32% converts all lowercase characters in a string to uppercase. This feature is valuable because some terminals transmit both forms of alphabetic characters. The lowercase characters are between CHR$(97) and CHR$(122); uppercase characters are between CHR$(65) and CHR$(90).

For example:

```
A$="abcde"

Ready

PRINT CVT$$(A$,32%)
ABCDE
```

The value 64% in the CVT$$ function lets you use parentheses instead of square bracket characters as delimiters of a project-programmer number.

For example:

```
A$="[200,245]"

Ready

PRINT CVT$$(A$,64%)
(200,245)
```

Use the value 64% when printing account numbers to terminals without square bracket characters. Most terminals have parenthesis characters, whereas many terminals lack square bracket characters.

The value 256% in the CVT$$ function prevents any alteration of characters inside quotation marks, except parity bit trimming—set by M% = 1%. Regardless of other values in the parameter M%, when 256% is included no operations are performed in the source string on characters inside quotation marks.

For example:

```
A$="abcde' fgh 'iJK"

Ready

PRINT CVT$$(A$,8%+32%+256%)
ABCDE' fgh 'IJK
```

Generally, the precedence of operations that BASIC–PLUS performs on the string is in increasing order of the values in the parameter M%. (The 256% value, however, is the exception; its precedence ranks between 1% and 2%.) This order implicitly determines which operations are performed on the string.

For example, consider the case where characters in the source string have their parity bit set, and you do not select the parity trimming option. Subsequent comparisons required by other options might not be successful, because the system compares source characters with ASCII characters that lack parity. A space (SP) character in no parity or odd parity form (CHR$(32)) does not equal a space (SP) character in even parity form (CHR$(160)).

In some text processing applications, the parity bit of each character functions as a flag instead of a parity bit. It is important in such cases to keep the parity bit in the input character of the string. BASIC–PLUS does not change or discard these flagged characters if you do not select the parity trimming option.

The precedence of operations affects the result of values given in the CVT$$ function. If you give the values 2% or 8% in the CVT$$ function, the values 8%, 16%, and 128% have no effect on the output string. Because the first option performed (2%) removes all space and tab characters from the string, the remaining values dealing with space and tab characters have no effect.

In like manner, the value 16% applies to all space and tab characters not discarded by the 2% and 8% options. Thus, to maintain at least a single space interval in a string, your program must give the 16% value and omit the 2% and 8% values.

The CVT$$ function often eliminates the need for special code in BASIC–PLUS programs to handle string input. For example, the following program manipulates an input string at lines 1020 through 1050.

```
LISTNH
10    EXTEND          ! USE BASIC-PLUS EXTEND MODE &

900   DIM IN.CHARS%(128%), OUT.CHARS%(128%) &
                ! DIMENSION TWO ARRAYS TO HANDLE THE BYTE REPRESENTATION &
                ! OF THE INPUT AND OUTPUT STRINGS. &

1000  PRINT "This program will remove all leading spaces and tabs" &
      \ PRINT "from a string, and convert all characters to uppercase." &
      \ PRINT &

1010  PRINT "Your string"; &
      \ INPUT LINE IN.STRING$ &

1015  ! &
      !         LINES 1020 - 1050 PERFORM THE CONVERSIONS &

1020  CHANGE IN.STRING$ TO IN.CHARS% &
                ! CONVERT THE INPUT STRING TO BYTES &
      \ IN.LEN% = IN.CHARS%(0%) &
                ! REMEMBER THE LENGTH OF THE INPUT STRING &
      \ OUT.PTR% = 0% &
                ! INITIALIZE THE OUTPUT CHARACTER POINTER &

1030  GOTO 1040 UNLESS IN.CHARS%(IN.PTR%)=9% OR IN.CHARS% (IN.PTR%)=32% &
         FOR IN.PTR%=1% TO IN.LEN% &
                ! SKIP OVER ALL LEADING TABS (9) AND SPACES (32), LEAVE &
                !  THIS LOOP WHEN WE ENCOUNTER A SIGNIFICANT CHARACTER &
                !  OR THE END OF THE STRING. &

1040  FOR IN.PTR% = IN.PTR% TO IN.LEN% &
                ! NOW COPY ANY REMAINING CHARACTERS IN THE INPUT STRING, &
                !  CONVERTING TO UPPERCASE, AS NECESSARY. &
      \         OUT.PTR% = OUT.PTR% + 1% &
                        ! POINT TO THE NEXT OUTPUT CHARACTER. &
      \         IN.CHAR% = IN.CHARS%(IN.PTR%) &
                        ! GET A CHARACTER FROM THE INPUT STRING. &
      \         IF IN.CHAR% >= 97% AND IN.CHAR% <= 122% THEN &
                        ! IF CHAR IS BETWEEN LOWERCASE "A" AND "Z" THEN &
                        OUT.CHARS%(OUT.PTR%) = IN.CHAR% - 32% &
                                ! MAKE THE CHARACTER UPPERCASE. NOTE &
                                !  THAT ASCII (LOCASE CHARACTER)-32 EQUALS &
                                !  ASCII (UPCASE CHARACTER). &

                ELSE    OUT.CHARS%(OUT.PTR%) = IN.CHAR% &
                                ! IF THE CHARACTER IS NOT LOWERCASE, THEN &
                                !  COPY IT DIRECTLY. &

1050  NEXT IN.PTR% &
                ! GO GET THE NEXT CHARACTER &
      \ OUT.CHARS%(0%) = OUT.PTR% &
                ! SET THE LENGTH OF THE NEW STRING. &
      \ CHANGE OUT.CHARS% TO OUT.STRING$ &
                ! CONVERT THE BYTE REPRESENTATION TO A STRING. &

1060  PRINT &
      \ PRINT "The old string was =>"; IN.STRING$; &
      \ PRINT "The new string is ==>"; OUT.STRING$; &
                ! PROVE THAT THE CONVERSION WORKED. NOTE THAT THE &
                !  CARRIAGE RETURN AND LINE FEED THAT THE USER TYPED &
                !  WILL BE PRINTED HERE, SINCE THEY ARE A PART OF THE &
                !  STRING. &

32767 END

Ready
```

```
RUNNH
This program will remove all leading spaces and tabs
from a string, and convert all characters to uppercase.

Your string?              ThIS STring WILl be fixeD

The old string was =>           ThIS STring WILl be fixeD
The new string is ==>THIS STRING WILL BE FIXED

Ready
```

You can replace lines 1020 through 1050 with a single CVT$$ function at line 1020, as shown in the following sample code. The value of 40% (32%+8%) in the CVT$$ function in line 1020 produces the same results as lines 1020 to 1050 in the original program.

```
LISTNH
10    EXTEND      ! USE BASIC-PLUS EXTEND MODE &

1000  PRINT "This program will remove all leading spaces and tabs" &
      \ PRINT "from a string, and convert all characters to uppercase." &
      \ PRINT &

1010  PRINT "Your string"; &
      \ INPUT LINE IN.STRING$ &

1015  ! &
      ! DO THE CONVERSION AT LINE 1020 &

1020  OUT.STRING$ = CVT$$(IN.STRING$, 32%+8%) &
                  ! CLEAN UP THE INPUT STRING: &
                  ! 32 = CONVERT TO UPPERCASE, AND &
                  ! 8  = DISCARD LEADING SPACES AND TABS &

1060  PRINT &
      \ PRINT "The old string was =>"; IN.STRING$; &
      \ PRINT "The new string is ==>"; OUT.STRING$; &
                  ! PROVE THAT THE CONVERSION WORKED. NOTE THAT THE &
                  !  CARRIAGE RETURN AND LINE FEED THAT THE USER TYPED &
                  !  WILL BE PRINTED HERE, SINCE THEY ARE A PART OF THE &
                  !  STRING. &

32767 END

Ready

RUNNH
This program will remove all leading spaces and tabs
from a string, and convert all characters to uppercase.

Your string?              conVert THis stringG TO ALL caps

The old string was =>           conVert THis stringG TO ALL caps
The new string is ==> CONVERT THIS STRING TO ALL CAPS

Ready
```

10.5.2 XLATE Function

The XLATE function translates a string from one storage code into another, using a translation table that you supply. For example, while reading a magnetic tape file, you may need to translate data from EBCDIC code to ASCII code so that it can be processed by the PDP–11.

The XLATE function has the form:

XLATE (<string1> ,<string2>)

The first argument, <string1> , is the *source* string; the second argument, <string2>, is the *table* string. XLATE returns a string value called the *target* string.

To perform the XLATE function, BASIC–PLUS operates sequentially on the characters in the source string. BASIC–PLUS uses the numeric value of each character (0 to 255) as an index into the table string. (Zero means the first character, 1 means the second, and so forth.)

For each character in the source string, BASIC–PLUS looks up the index value in the table string. If the selected character in the table string is not zero and if the index value is in the table, BASIC–PLUS appends the character value from the table string to the target string. BASIC–PLUS does not transfer any character to the target string if these two conditions are not met. Thus, the target string is equal to or shorter than the source string. In addition, the target string cannot contain null characters (ASCII code 0).

For example, suppose the first character in the source string is an uppercase A (ASCII code 65). BASIC–PLUS looks at the sixty-fifth character in the table string. If it is an uppercase A, BASIC–PLUS transfers an A to the first position in the target string. If it is another character (for example, an asterisk), BASIC–PLUS transfers that character into the first position in the target string. BASIC–PLUS does not transfer any character into the first position of the target string if the sixty-fifth character in the table string is a zero or if the table string contains fewer than 65 characters.

The following program uses the XLATE function to translate all lowercase letters in a string to uppercase and to remove characters other than letters or digits. Uppercase letters and digits do not change.

```
50 EXTEND
100 ! Build the table string.  (For each place that a zero &
    ! appears in the table string, the character in that   &
    ! position will not appear in the target string.)      &
    !
200 TABLE$ = STRING$(48%,0%)     !Do not translate CHR$(0%) through "/" &
           + "0123456789"        !Keep digits the same &
           + STRING$(7%,0%)      !Do not translate ":" through "@" &
           + "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  !Keep uppercase letters the same&
           + STRING$(6%,0%)      !Do not translate "[" through "`" &
           + "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  !Translate lowercase letters &
                                           !to uppercase &
```

(continued on next page)

```
    !Table ends here.  Characters "{" through <RUBOUT> (CHR$(127%) &
    !will be left out of target string by default. &
    !
300 PRINT &
    \ INPUT "String to translate ";TEST.STRING$    !Get user's test string &
    \ GOTO 32767 UNLESS LEN(TEST.STRING$) <> 0%    !Quit if user types <RETURN> &

400 TRANSLATED.STRING$ = XLATE(TEST.STRING$,TABLE$)  !Translate test string &

500 PRINT "The string you entered translated to ";TRANSLATED.STRING$; &
    \PRINT &
    \REMOVED.COUNT% = LEN(TEST.STRING$) - LEN(TRANSLATED.STRING$) &
    \PRINT CHR$(7%);REMOVED.COUNT%;" error characters removed"; &
     IF REMOVED.COUNT% <> 0% &
    \PRINT &
    \GOTO 300
32767 END


RUNNH
String to translate? abcdefg*$.123ABC
The string you entered translated to ABCDEFG123ABC
 3 error characters removed
```

10.6 User-Defined String Functions

You can define your own string functions with the DEF* statement. Write string functions like numeric functions (see Sections 9.9 and 13.1). Indicate the function as a string function with a dollar sign ($) after the function name.

User-defined string functions return string values. For example, the following multi-line function returns the string that comes first in alphabetical order:

```
100    DEF* FNF$(A$,B$) &
       \ FNF$=A$ &
       \ IF A$>B$ THEN FNF$=B$
110    FNEND
```

See Section 13.1 for more information about multi-line functions.

The following function combines two strings into one string:

```
10    DEF* FNC$(X$,Y$)=X$+Y$
```

String functions can have both numeric and string arguments. However, you cannot use numbers as arguments in a function where strings are expected or vice versa. Line 80 is unacceptable because FNA$ is defined to have a string argument:

```
10    DEF* FNA$(A$) = CHR$(LEN(A$)+1)
80    LET Z=FNA$(4)
```

BASIC–PLUS prints:

```
?Arguments don't match at line 80
```

A function can have arguments of mixed types. BASIC–PLUS converts integers to floating-point numbers (and vice versa) when it needs an argument. Integer and floating-point values, however, are not interchangeable with strings. (See Chapter 11 for information on integer and floating-point operations.)

The following example defines a function with a string argument, a floating-point argument, and an integer argument:

```
DEF* FNA$(A$,B%,C)=A$+"("+NUM1$(B%)+") is "+NUM1$(C))
```

You can call this function with either of these statements:

```
X1$ = FNA$ (A$, P,  3%)
X1$ = FNA$ (A$, P%, 3 )
```

By comparison, the next statement is incorrect because the arguments are not in the right order:

```
X1$ = FNA$ (P,  A$, 3)
```

The following code is a string function that returns the leftmost five characters from the sum of three arguments:

```
LISTNH
75    DEF* FNA$(X,Y,Z) = LEFT (NUM$(X+Y+Z),5)
80    PRINT FNA$(100,20,3)
32767 END

Ready

RUNNH
 123

Ready
```

NUM$(123) is the five-character string:

"(space)123(space)"

10.7 String Arithmetic

The optional string arithmetic feature consists of six functions that treat strings of numeric characters as arithmetic operands. String arithmetic offers greater arithmetic precision with large numbers and fractions than floating-point arithmetic. Thus, it eliminates the need for scaling.

Table 10–2 describes the six functions that make up the optional string arithmetic feature: SUM$, DIF$, PROD$, QUO$, PLACE$, and COMP%.

The arguments A$ and B$ in these functions can be string constants, string variable names, or string expressions. Specify strings that consist of numeric characters with an optional leading sign and an optional decimal point. A$ and B$ can each be up to 56 characters long, including the plus or minus sign and the decimal point.

Remember to end numeric string variable names with a dollar sign ($) and to enclose numeric string constants in single or double quotation marks.

The P% argument is an integer expression that specifies the level of precision. See Section 10.7.1 for more information.

Table 10–2: Optional String Arithmetic Functions

Function Code	Meaning

SUM$(A$,B$)	Yields the arithmetic sum A$ + B$ of numeric strings A$ and B$. For example:
	`S1$='12349.1789'`
	`Ready`
	`PRINT SUM$(S1$,'89.4545454545')`
	`12438.6334454545`

DIF$(A$,B$)	Yields the arithmetic difference, A$–B$, of numeric strings A$ and B$. For example:
	`B$='9876.54321'`
	`Ready`
	`PRINT DIF$(B$,'78.89')`
	`9797.65321`

PROD$(A$,B$,P%)	Yields the product, A$ times B$, rounding to P places. For example:
	`A$='12345.6789'`
	`Ready`
	`B$='9876.54321'`
	`Ready`
	`PRINT PROD$(A$,B$,6)`
	`121932631.112635`

QUO$(A$,B$,P%)	Yields the quotient, A$ divided by B$, with rounding to P places. For example:
	`C$='3.5'`
	`Ready`
	`V9$=QUO$(C$,'1.7777',3)`
	`Ready`
	`PRINT V9$`
	`1.969`

(continued on next page)

Table 10–2: Optional String Arithmetic Functions (Cont.)

Function Code	Meaning

PLACE$(A$,P%)	Rounds A$ to P places. For example:
	`A$='12345.6789'`
	`Ready`
	`PRINT PLACE$(A$,3)`
	`12345.679`

COMP%(A$,B$)	Yields a truth value based on the result of a numeric comparison:
	–1 if A$ < B$
	0 if A$ = B$
	1 if A$ > B$
	For example, if A$ has the same value as in the previous example and A1$ is equal to PLACE$(A$,3):
	`PRINT COMP%(A$,A1$)`
	`-1`
	`Ready`
	`PRINT COMP%(A1$,A$)`
	`1`

10.7.1 String Arithmetic Precision

In SUM$ and DIF$, the precision of the larger argument determines the precision of the result. For example:

```
10 A$ = "89516.332948661"
20 B$ = "3602.59619"
30 PRINT SUM$(A$,B$)
32767 END
RUNNH
93118.929138661
```

The result is expressed to the precision of A$, the more precise argument.

```
10 A$ = "89516.332948661"
20 B$ = "3602.59619"
30 PRINT DIF$(A$,B$)
32767 END
RUNNH
85913.736758661
```

The result is expressed to the precision of A$, the more precise argument.

The PROD$, QUO$, and PLACE$ functions let you specify the level of arithmetic precision you want in their results. You specify the level of precision with the P% argument.

P% is an integer expression. It can be positive or negative. A positive P% value less than 5000 rounds the result to P significant digits to the right of the decimal point. For example:

```
LISTNH
100      EXTEND
         !ALLOWS LONG VARIABLE NAMES
110      INPUT 'ENTER TWO NUMERIC STRINGS TO BE MULTIPLIED';&
                STRING.A$,STRING.B$ &
         \ INPUT 'TO HOW MANY DECIMAL PLACES';PNO%
120      PR$=PROD$(STRING.A$,STRING.B$,PNO%) &
         \ PRINT 'ANSWER IS  ';PR$
32767    END

Ready

RUNNH
ENTER TWO NUMERIC STRINGS TO BE MULTIPLIED? 56453.346, '879.0004532'
TO HOW MANY DECIMAL PLACES? 12
ANSWER IS  49622516.718654072

Ready

RUNNH
ENTER TWO NUMERIC STRINGS TO BE MULTIPLIED? .00009067543,0134.2340345
TO HOW MANY DECIMAL PLACES? 10
ANSWER IS  .0121717288

Ready
```

You can use a negative P% value if you want an approximate result. For example:

```
100      PRINT 'ENTER A LARGE NUMBER' &
         \INPUT A$ &
         \IF A$ = '0' THEN 150
110      LET B$ = SUM$(A$,B$) &
         ! ACCUMULATE TOTAL OF INPUT STRINGS
120      GO TO 100
150      B$ = PLACE$(B$,-6%)
160      PRINT 'TOTAL IS APPROXIMATELY '; B$; ' MILLION'
32767    END
```

You can also use the following statement in place of lines 150 and 160:

```
150    PRINT 'TOTAL IS APPROXIMATELY ';PLACE$(B$,-6%); ' MILLION'
```

To truncate a result instead of rounding it, specify P% in the form:

P% + 10000%

For example, in the following PROD$ function, a P% value of 10012% truncates the result to 12 significant digits to the right of the decimal point:

```
10 A$ = "89516.332948661"
20 B$ = "3602.59619"
30 P% = 10012%
40 PRINT PROD$(A$,B$,P%)
32767 END
RUNNH
322491200.023617584201
```

When P% is negative, the result of P% + 10000% can be in the range 0 to 10000. To distinguish between a large value of P and truncation after scaling, BASIC–PLUS compares P% to 5000. The result is rounded if P% is less than 5000. Otherwise, the result is divided by 10^P and truncated.

Negative P% causes the result to be effectively divided by 10^P and rounded P places to the left of the decimal point.

To illustrate, the following examples compare the results you get when you specify positive and negative P% values with the same arguments:

```
10 A$ = "89516.332948661"
20 B$ = "3602.59619"
30 P% = 14%
40 PRINT PROD$(A$,B$,P%)
32767 END
RUNNH
322491200.02361758420159


10 A$ = "89516.332948661"
20 B$ = "3602.59619"
30 P% = -1%
40 PRINT PROD$(A$,B$,P%)
32767 END
RUNNH
32249120


10 A$ = "89516.332948661"
20 P% = 4%
30 PRINT PLACE$(A$,P%)
32767 END
RUNNH
89516.3329


10 A$ = "89516.332948661"
20 P% = -3%
40 PRINT PLACE$(A$,P%)
32767 END
RUNNH
90
```

The following examples show results obtained for P% values between 5000 and 10000:

```
10 A$ = "89516.332948661"
20 B$ = "3602.59619"
30 P% = 5500%
40 PRINT QUO$(A$,B$,P%)
32767 END
RUNNH
0


10 A$ = "89516.332948661"
20 P% = 9997%
30 PRINT PLACE$(A$,P%)
32767 END
RUNNH
89
```

If you specify a value outside the range for P%, you receive the error message "%Floating point error at line n." BASIC–PLUS returns the result with as much precision as possible. For example:

```
100 A$ = '2'
200 B$ = '3'
300 PRINT QUO$(A$,B$,60)
32767 END
RUNNH
%Floating point error at line 300

.666666666666666666666666666666666666666666666666666666
```

Table 10–3 summarizes the values of P% and their effects on Y$ (the value returned by PROD$, QUO$ and PLACE$).

Table 10–3: Precision Values in PROD$, QUO$, and PLACE$ Functions

Value of P%	Effect on Y$
0 < = P% < 5000	Y$ is rounded P significant digits to the right of the decimal point.
P% = P% + 10000	Y$ is truncated P significant digits to the right of the decimal point.
P% < 0	Y$ is divided by 10^P and rounded.
5000 < = P% < 10000	Y$ is divided by 10^P and truncated.

10.7.2 Combining String Arithmetic Functions

You can nest string arithmetic functions (as operands in other string arithmetic functions, for example) to specify complicated arithmetic or algebraic operations. For example, consider the following LET statement:

```
X = A*B + C/D
```

You can write this in string arithmetic as:

```
X$ = SUM$(PROD$(A$,B$,10), QUO$(C$,D$,10))
```

The following statement is legal, but it concatenates the product and quotient strings instead of summing them:

```
X$ = PROD$(A$,B$,10) + QUO$(C$,D$,10)
```

The following program illustrates these two ways of combining string functions:

```
LISTNH
100    A$="12." !EVALUATE (12*3.3) + (10/4)
110    B$="3.3"
120    C$="10."
130    D$="4."
190    !
200    X$=SUM$ (PROD$(A$,B$,10.), QUO$(C$,D$,10.)) !THE RIGHT WAY
210    PRINT "DOING IT THE RIGHT WAY, X$="; X$
220    !
300    X$=PROD$(A$,B$,10.) + QUO$(C$,D$,10.) !THE WRONG WAY
310    PRINT "DOING IT THE WRONG WAY, X$="; X$
320    !
32767 END

Ready

RUNNH
DOING IT THE RIGHT WAY, X$=42.1
DOING IT THE WRONG WAY, X$=39.62.5

Ready
```

Chapter 11
Integer and Floating-Point Operations

BASIC–PLUS has two numeric data types: floating-point numbers and integers. Floating-point numbers can have fractional parts; integers are whole numbers. This chapter describes the operations you can perform using these two data types.

Topics include:

- Integer arithmetic
- Logical operations on integers
- Floating-point and scaled arithmetic
- Mixed-mode arithmetic

11.1 Introduction to Integers

BASIC–PLUS stores integer data in binary form. Each integer uses one PDP–11 16–bit word of memory.

Integers have two characteristics you can take advantage of in BASIC–PLUS programming:

1. They are numbers.
2. They are bit patterns.

As numbers, integers can range in value from –32768% to 32767%. Integers are useful as counters in program loops and other repetitive operations. Your programs will be smaller and more efficient if you use integers for arithmetic operations that involve whole numbers in the range –32768% to 32767%. Integers use less space than floating-point numbers, and integer arithmetic is faster than floating-point arithmetic. For information on integer arithmetic, see Sections 11.3 through 11.6 of this chapter.

Like everything else inside a computer, an integer is a bit pattern. Each integer value corresponds to a unique sequence of bits that have either the value 1 or 0. A bit whose value is 1 is "on" or "set"; a bit whose value is 0 is "off" or "clear."

BASIC–PLUS uses integers as bit patterns to perform logical operations (such as determining whether a logical expression is true or false). In addition, BASIC–PLUS gives you access to this process as a programming tool. By performing logical operations on integers, you can test and manipulate individual bits inside a PDP–11 16–bit word. Section 11.7 describes the technique, which has several useful applications in BASIC–PLUS programming.

Whether you use integers as numbers or as bit patterns, you always specify them to BASIC–PLUS as positive or negative decimal numbers, and BASIC–PLUS always returns them to you in the same form.

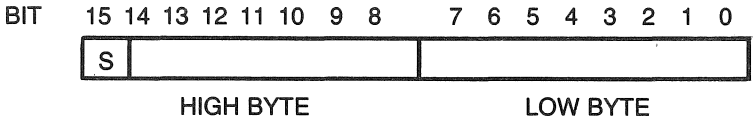
While working with integers as decimal numbers is natural when you use them to do arithmetic, you need to do some "translation" when you use them as bit patterns. The next section gives you the information you need to make that translation. It also explains why integer arithmetic in BASIC–PLUS works the way it does.

11.2 Internal Integer Format

This section explains how an integer is stored in memory. If you plan to use integers only in arithmetic operations, you may want to skip this section for now. But if you plan to use integers in logical operations, read this section before you read Section 11.7.

Figure 11–1 shows the internal format of an integer.

Figure 11–1: Internal Integer Format



The first 15 bits (0 through 14) each correspond to a positive power of 2. For example, a value of 1 in bit 0 means 1% (2^0), a value of 1 in bit 3 means 8% (2^3), and a value of 1 in bit 8 means 512% (2^8).

Bit 15, the leftmost bit, has a special purpose: it stores a number's sign. Thus, it is called the "sign bit." A value of 0 in the sign bit means a number is positive; a value of 1 in the sign bit means a number is negative.

Bits 0 through 7 are called the "low byte" or the "low order bits"; bits 8 through 15 are called the "high byte" or the "high order bits." (You will encounter these terms later in this manual; they are also used in the *RSTS/E Programming Manual.*) The low byte can store a value in the range 0% through 255%; the high byte can store a value in the range 512% through 32512%.

A word that contains values of 1 in bits 0 through 14 (and 0 in bit 15) stores the value 32767%, the largest value an integer can have in BASIC–PLUS. A word with all bits set to 0 stores the value 0%.

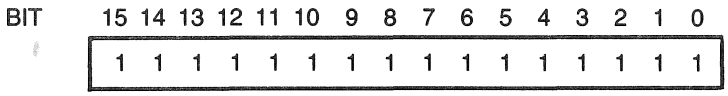
Each integer value has a unique bit pattern. To find the value of an integer from its bit pattern, you add the powers of 2 that correspond to each of the "1" bits. For example, consider a word where bits 0, 3, and 8 are 1 and the rest of the bits in the word are 0. The bit pattern is:

0000000100001001

The integer value stored in this word is 1%+8%+512%, or 521%. The number is positive because the sign bit is 0.

You use the same process for negative numbers. The key to deciphering negative numbers is the sign bit. The sign bit has value as well as sign; its value is –32768%.

The following example shows how to determine the value of a negative number from its bit pattern. The internal format for the integer value –1% is:



The integer value –1% has "all bits on." To see why, add the values of all the bits. Bit 15 has the value –32768%; the rest of the bits are positive powers of 2 and add up to 32767%. The result is –1%.

The sign bit has the value –32768% because the PDP–11 computer does calculations using numbers in their "two's complement" form. (A binary number's "two's complement" is its value with all the bits reversed, plus 1.) This method allows arithmetic to be done inside the machine as a continuous operation, regardless of whether numbers are positive or negative.

Two's complement arithmetic has the following characteristics:

- There are 32,768 positive integers: 0% to 32767%.
- There are 32,768 negative integers: –1% to –32768%.
- When 1 is added to the highest number in the range, the result is the lowest number. For example, 32767% + 1% yields –32768%, the value of the sign bit. (You can use this expression in certain statements to set a word's sign bit. Setting the sign bit causes the system to perform special functions. See Section 14.5.3.1 for an example.)

The next four sections describe the arithmetic operations you can perform using integers as whole numbers. In Section 11.7, which describes logical operations on integers, you will see how to use integers as bit patterns.

11.3 Integer Constants and Variables

Numeric variables and constants without a % suffix are stored internally as floating-point numbers. When operations deal with whole numbers, you can achieve significant economies in storage space and execution time with the integer data type, which uses only one computer word per value. Integer arithmetic is also significantly faster than floating-point arithmetic. Integer variables can assume values in the range –32768% to +32767%. Integer constants can assume values in the range –32767% to +32767%.

You specify that a constant, variable, or function is to be an integer by ending its name with the % character. For example:

```
A%      100%    FNX%(Y)
A1%       4%    FNL%(N%,L%)
```

When you assign a floating-point value to an integer variable, the fractional portion of that number is lost. The number is not rounded to the nearest integer value. (A FIX function is performed rather than an INT function; see Section 9.8.) For example:

```
A% = 1.1

A% = 1.9
```

Either of these would assign the value 1 to A%.

11.4 Integer Arithmetic

In integer arithmetic, the number range –32768 to +32767 is treated as continuous, with the number after +32767 equal to –32768. But adding large positive numbers can result in unexpected negative numbers, as in 32767% + 2% = –32767%.

Integer division forces truncation of any remainder; for example 5%/7% = 0 and 199%/100% = 1. Integer and floating-point data can be freely mixed in such operations (see Section 11.9). The answer is stored in the format indicated as the resulting variable. For example:

```
125   LET X% = N% + FNA(R)*2.
```

BASIC–PLUS evaluates the expression to the right of the equal sign and then truncates the result to provide an integer value for X%.

Where program size is critical, use the % character to generate integer values because it uses significantly less storage space. For example:

```
120   FOR I% = 1% TO 10%
```

This statement takes less storage space and executes faster than:

```
120   FOR I = 1 TO 10
```

11.5 Integer I/O

Input and output of integer variables is performed in exactly the same manner as input and output on floating-point variables. (Remember that where a floating-point variable has an integer value, BASIC–PLUS automatically prints it as an integer. The value is still stored internally as a floating-point number, however, thus taking more storage space.)

It is illegal to provide a floating-point value for an integer variable through either a READ or INPUT statement. For example:

```
LISTNH
110    READ A% &
       \ PRINT A%
600    DATA 2.7
32767 END

Ready

RUNNH
%Data format error at line 110

Ready
```

Change line 600 to:

```
600    DATA 3
```

The result is:

```
RUNNH
 3

Ready
```

11.6 User-Defined Integer Functions

You can write functions to handle integer variables as well as floating-point variables. You define an integer function by ending its name with the % character.

For example, you can define a function that returns the remainder when one integer is divided by another:

```
110    DEF FNR%(I%,J%) = I%-((I%/J%)*J%)
```

You can call this function later in the program as:

```
200    PRINT FNR%(A%,11%)
```

You can use integer arguments where floating-point arguments are expected and vice versa. BASIC–PLUS performs the necessary conversions. You cannot, however, use strings and numbers interchangeably.

```
75    DEF FNA%(X%) = X% - 1%
80    LET Z% = FNA%(12.34)
```

This example is acceptable. Z equals 11 after line 80 has been executed.

11.7 Logical Operations on Integer Data

So far, you have learned how to use integers as whole numbers in mathematical calculations. You can also use integers as logical values in BASIC–PLUS.

BASIC–PLUS produces logical values when it evaluates expressions that contain:

- Relational operators, such as = and <
- Logical operators, such as AND, OR, and XOR

These operators cause BASIC–PLUS to compare two operands and store the result of the comparison in a one-word integer variable. This result is a logical value.

11.7.1 The Logical Values –1% and 0%

If you have written programs with IF–THEN statements, you have already used the logical values –1% and 0%. You know these values as "true" and "false."

BASIC–PLUS uses –1% (all bits on) to mean "true" and 0% (all bits off) to mean "false." Two types of expressions always return one of these two values:

- Relational expressions
- Logical expressions that compare two relational expressions

11.7.1.1 Relational Expressions

In evaluating a relational expression, BASIC–PLUS compares two numbers or two string values. The comparison has two possible results: the logical value –1% (true) or the logical value 0% (false). BASIC–PLUS stores the result in an integer variable, in one PDP–11 word of memory. For example:

```
15 IF A% > B% THEN GOTO 100
```

BASIC–PLUS performs two steps when it executes this statement:

1. It compares the current values of A% and B% and stores the result (–1% or 0%) in an integer variable.
2. It tests the value of this integer variable. If its value is 0%, BASIC–PLUS executes the next line in the program; if it is –1% (or any other nonzero value), BASIC–PLUS transfers control to line 100.

You can program this operation as two separate steps. You can store a logical value in an integer variable at one point in a program and perform one or more logical tests on this stored integer at another point in the program.

For example:

```
20 X% = (A% > B%)
 .
 .
 .
50 IF X% THEN GOTO 100
```

When line 20 is executed, BASIC–PLUS evaluates A% > B% and stores the logical value –1% or 0% in the integer variable X%. When line 50 is executed, BASIC–PLUS tests the current value of X% to determine if it is true or false and proceeds accordingly.

11.7.1.2 Logical Expressions

BASIC–PLUS also evaluates as –1% or 0% a logical expression that compares two relational expressions. For example:

```
20 IF (A > B%) AND (C% < D%) THEN GOTO 4000
```

To evaluate this logical expression, BASIC–PLUS first evaluates each relational expression. For each relational expression, BASIC–PLUS stores a logical value of either –1% or 0% in an integer variable. Next BASIC–PLUS compares the contents of these two integer variables based on the logical operator AND. The result of the AND operation is a single logical value of either –1% (true) or 0% (false). BASIC–PLUS uses this logical value to determine whether to execute line 4000.

You can store the result of a logical expression in one statement and test it in another statement:

```
20 X% = (A% < B%) AND (C% > D%)
 .
 .
 .
50 IF X% THEN GOTO 5000
```

As before, BASIC–PLUS interprets 0% as false and –1% or any other nonzero value as true.

11.7.2 Other Logical Values

Logical values are not restricted to 0% or –1%; every integer value is a logical value. Thus, you can store any integer value in a variable and test it at a later time.

For example:

```
20 X% = 5%
 .
 .
 .
50 IF X% THEN GOTO 1000
```

Since 5% is interpreted as true, control goes to line 1000. Line 20 can also contain an integer expression. For example:

```
20 X% = (A% + B%)/ 2%
```

You can use any expression or function that yields an integer value in an IF–THEN statement. For example:

```
30 IF LEN(TYPED.RESPONSE$) THEN GO TO 2000
```

This statement, which contains the string function LEN, transfers control to line 2000 if there are characters in the string variable TYPED.RESPONSE$. If TYPED.RESPONSE$ contains no characters, BASIC–PLUS executes the next line in the program.

11.7.3 How BASIC–PLUS Performs Logical Operations

Up to now, you have used logical values to mean true or false. But logical values have another use in BASIC–PLUS programming. You can use the logical operators to examine, set, and clear individual bits inside PDP–11 16–bit words. To see how to use logical operators for this purpose, you need to know more about how BASIC–PLUS performs logical operations.

The BASIC–PLUS logical operators (AND, OR, NOT, XOR, IMP, and EQV) act on entire 16–bit PDP–11 words. A logical operation is the result of 16 single-bit logical operations.

BASIC–PLUS performs a logical operation by comparing the bit patterns of two integers, one bit at a time. The type of comparison that BASIC–PLUS makes for each pair of bits depends on the logical operator that you specify.

The truth tables in Table 11–1 show, for each logical operator, how BASIC–PLUS compares bit pairs in two integers.

For each operator in this table, A is a bit in one integer value, and B is the bit that occupies the same position (the "matching" bit) in the other integer value. The result of the logical comparison, shown on a third line, is stored in a third integer value. The NOT operator, unlike the rest, has only one operand.

Table 11–1: Truth Values for Logical Operations

AND

A	1	1	0	0
B	1	0	1	0
A AND B	1	0	0	0

OR

A	1	1	0	0
B	1	0	1	0
A OR B	1	1	1	0

XOR

A	1	1	0	0
B	1	0	1	0
A XOR B	0	1	1	0

EQV

A	1	1	0	0
B	1	0	1	0
A EQV B	1	0	0	1

IMP

A	1	1	0	0
B	1	0	1	0
A IMP B	1	0	1	1

NOT

A	1	0
NOT A	0	1

The following examples show how BASIC–PLUS produces logical values. Each example compares the integers 85% and 28% using a different logical operator.

The first example prints the *logical product* of two integers, 85% and 28%, in immediate mode. (A logical product is the result of an AND operation.) An AND operation yields a 1 only when matching bits are both 1 and clears bits that are not set in both values. You can use an AND operation to see if two values have any set bits in common, to test specific bits, or to "mask" certain bits (see Section 11.7.4.3).

```
PRINT 85% AND 28%
 20

Ready
```

To find the logical product of 85% and 28%, BASIC–PLUS performs 16 AND operations, one on each pair of matching bits in the integers 85% and 28%. BASIC–PLUS stores the result of these 16 AND operations in a third integer value. The following diagram shows the bit pattern of 85%, 28%, and the result.

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
85% =	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1
28% =	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0
85% AND 28% =	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0

The bit pattern in the result is the integer value 20% (16% + 4%). This result occurs because the 16% bit and the 4% bit are set in both values being compared.

The second example prints the *logical sum* of 85% and 28%. (The logical sum is the result of an OR operation.) Because an OR operation yields a 1 if either or both bits in a position are 1, you can use the OR operator to set specific bits in a word.

```
PRINT 85% OR 28%
 93

Ready
```

To find the logical sum of 85% and 28%, BASIC–PLUS performs 16 OR operations, one on each pair of matching bits in 85% and 28%. The following diagram shows this operation:

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
85% =	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1
28% =	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0
85% OR 28% =	0	0	0	0	0	0	0	0	0	1	0	1	1	1	0	1

The logical sum of 85% and 28% is 93% (64% + 16% + 8% + 4% + 1%). The OR operator sets a bit if one or both bits being compared are set.

The third example prints the *logical difference* of 85% and 28%. (The logical difference is the result of an XOR operation.) An XOR operation yields a 1 only if either bit in a matching pair is set; otherwise it yields a 0. You can use XOR to toggle specific bits in a word. ("Toggling" a bit means setting a 0 bit to 1 or a 1 bit to 0.)

```
PRINT 85% XOR 28%
 73

Ready
```

To find the logical difference, BASIC–PLUS performs 16 XOR operations. The following diagram shows the result:

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
85% =	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1
28% =	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0
85% XOR 28% =	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1

The logical difference of 85% and 28% is 73% (64% + 8% + 1%).

11.7.4 Programming Applications

Each bit in a PDP–11 word can store a piece of information. A set (1) bit can indicate the presence of a condition; a clear (0) bit can indicate the absence of a condition. Thus, a single word in memory can store up to 16 pieces of information.

Both BASIC–PLUS and RSTS/E take advantage of this compact form of information storage. For example, when you open a device with the OPEN statement, BASIC–PLUS places several pieces of information about the device in the special variable STATUS. By testing bits in the STATUS variable, you can get information about the device you just opened. (Section 17.1.1 describes the STATUS variable and shows you how to test the bits.)

SYS system function calls let you communicate with the RSTS/E monitor. In some calls, the monitor returns information to your program in string form. Certain bytes in the string contain several "bits" of information. To read this information, you can convert the string to an integer and then use logical operations to perform bit tests. (The *RSTS/E Programming Manual* describes how to perform the conversion and test the bits.)

You can also use this technique in your own programs. You can use an integer variable to store several pieces of information. Then, at various points in a program, you can perform logical operations on that variable to test, set, and clear specific bits.

The rest of this section shows some ways to use logical operations.

11.7.4.1 Bit Tests

The AND operator performs a bit test. You can use AND on an integer variable and a power of 2 to see if a specific bit is on in a word. For example:

```
20 IF A% AND 8% THEN PRINT "Bit 3 is set" &
   ELSE PRINT "Bit 3 is clear"
```

This is the technique you use to test bits in the STATUS variable and in "flag words" of data returned by certain SYS calls. (See the *RSTS/E Programming Manual.*)

11.7.4.2 Setting or Clearing Bits

The following statements set and clear bit 7 in the integer variable FLAG%:

```
250 FLAG% = FLAG% OR 128%          !Turn on bit 7

260 FLAG% = FLAG% XOR 128%         !Turn it off if it is on;
                                   !Turn it on if it is off

270 FLAG% = FLAG% AND (NOT 128%) !Turn it off unconditionally
```

11.7.4.3 Bit Masks

Sometimes you may be interested in some but not all of the information stored in a word of memory. You can use an AND operation to hide or "mask" part of a bit pattern so you can access the part you are interested in. For example:

```
30 L% = W% AND 255%
```

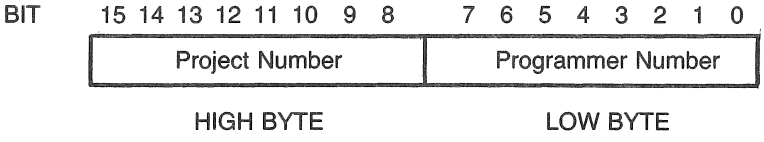
This statement masks out the high order bits (8–15) of W%. The value 255% contains 1s in bits 0 through 7 and 0s in bits 8 through 15. Because an AND operation yields a 1 only if both bits are 1, the integer L% contains 0s in bits 8 through 15 and the same bit pattern as W% in bits 0 through 7. The following diagram shows this operation (W% is an arbitrary bit pattern):

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
W% =	0	1	0	0	1	1	0	1	0	1	0	1	0	1	0	1
255% =	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
W% AND 255% =	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1

(stored in L%)

The result shows that L% has the same bit pattern as W% in the low order bits and all 0s in the high order bits. The value 255% is a "bit mask" to hide the high order bits of a word.

One application of bit masking is reading data returned by the file name string scan SYS call, which is described in the *RSTS/E Programming Manual*. For example, the monitor returns the project-programmer number of the file in one word, as shown:



The data comes back in string form, so you must first convert the string data to integer data. (You can do this with the CVT$% function; see Section 17.5.1.) Assume that you do the conversion and you store the contents of this word in the integer variable PPN%. You can then read either half of the word by using 255% as a bit mask. For example:

```
PROGRAMMER% = PPN% AND 255%
PROJECT% = SWAP%(PPN%) AND 255%
```

The SWAP% function swaps the two bytes in the word. Using SWAP% to put the project number in the low byte lets you read the project number using the same bit mask you used for the programmer number. See Section 17.5.2 for information on the SWAP% function. For information on using system function calls, see the *RSTS/E Programming Manual*.

Here is another example of bit masking:

```
PRINT CHR$(ASCII("a") AND (NOT (32%))
A

Ready
```

The ASCII value of a lowercase letter and its uppercase form differ by 32. A lowercase value has bit 5 on; an uppercase value has bit 5 off. Except for this bit, an uppercase letter and a lowercase letter have the same value.

You convert a letter from its lowercase form to its uppercase form by clearing bit 5. The value NOT 32% is a "mask" to clear bit 5 because the value NOT 32% has all bits on except bit 5.

The CVT$$ function performs this operation as part of its function that converts letters to uppercase.

11.8 Floating-Point Arithmetic

Floating-point numbers occupy either two or four 16–bit words of storage in memory. BASIC–PLUS uses two words with the single-precision math package and four words with the double-precision math package. Section F.4 describes the internal format of the two math packages.

With the two-word format, you can accurately represent numbers up to six decimal digits. The four-word format lets you represent numbers up to 15 decimal digits. Both formats allow numbers in the range 10^–38 to 10^38, approximately. An attempt to assign or compute a number outside the allowed range causes the "%Floating point error" (ERR = 48).

BASIC–PLUS prints numeric results of floating-point calculations in decimal or exponential format, as described in Section 8.3. To print numbers larger than six digits, you can tailor the format with the PRINT–USING statement (see Section 15.3.2).

Usually you cannot represent fractional numbers exactly in binary notation. Also, certain calculations in floating-point result in an accumulated error. The following calculation, run in standard four-word floating-point, results in an accumulated error. The error results because the floating-point fraction .01 cannot be represented internally as that precise value.

```
LISTNH
100    X = 0.
110    X = X + .01 FOR I% = 1% TO 10000%
120    PRINT NUM1$(X-100.)
32767 END

Ready

RUNNH
-.000000000001776356839400 25

Ready
```

If no accumulated error exists, the result is 0. Running the example on a system using the two-word format generates a much greater accumulated error (approximately .00295).

NOTE

The scaled arithmetic feature, available only on systems with the double-precision math package, lets you avoid or reduce accumulated error. Section 11.10 describes scaled arithmetic.

11.9 Mixed Mode Arithmetic

You can perform arithmetic operations using a mix of integer and floating-point numbers. To force a floating-point representation of an integer constant, end it with a decimal point. To force an integer representation of a constant, end it with the % character. Constants without a decimal point or % character are termed *ambiguous*. The rest of this section describes the results of arithmetic operations using a mixture of integer and floating-point numbers.

If both operands of an arithmetic operation are explicitly either integer or floating-point, BASIC–PLUS generates integer or floating-point results, respectively. If one operand of an arithmetic operation is an integer and another is floating-point, BASIC–PLUS converts the integer to a floating-point representation and generates a floating-point result. For example:

```
PRINT 1%/2%;  1./2.;  1%/2.;  1./2%
 0  .5  .5  .5

Ready
```

In the first two operations, BASIC–PLUS generates the explicit results; in the second two, BASIC–PLUS converts the explicit integer and generates floating-point results.

Sometimes an ambiguous constant appears in an arithmetic expression (for example, 10 as opposed to 10% or 10.). If an integer variable or constant occurs anywhere to the left of the ambiguous constant, BASIC–PLUS represents it in integer format. Otherwise, BASIC–PLUS treats the ambiguous constant as a floating-point number. For example:

```
PRINT 1%/2;  1/2%;  1/2
 0   .5   .5

Ready
```

In the first operation, BASIC–PLUS treats the "2" as an integer. It does so because an explicit integer representation appears to the left in the expression. In the next two operations BASIC–PLUS treats the ambiguous constants as floating-point numbers. It does so because no explicit integer variable or constant appears to the left of the ambiguous constant in the expression.

Because formatting determines the results of many operations, you should explicitly impose the correct format with the percent sign or the decimal point. Compare the following calculations, assuming A%(2%)=0 in each expression:

```
PRINT A%(2%) + (32767+2)% A%(2%) + (32767.+2)
-32767   32769

Ready
```

The percent sign in the first expression forces an integer result. The decimal point in the second expression forces a floating-point result. The same principle applies in the following example:

```
PRINT 1% + 1/2% 1. + 1/2% 1 + 1/2
1   1.5   1.5

Ready
```

Use of an explicit percent sign or decimal point lets you control the result in mixed-mode operations.

11.10 Scaled Arithmetic

On a system with the double-precision floating-point (four-word) math package, the scaled-arithmetic feature lets you avoid or reduce accumulated error in the fractional part of a number. Systems with two-word precision do not have scaled arithmetic. To work with the scaled arithmetic feature, you use the SCALE command to specify a scale factor.

NOTE

The optional string arithmetic feature provides another alternative to scaled arithmetic for providing better precision in non-integer arithmetic. Unlike scaled arithmetic, which is limited to a scale factor between 1 and 6, string arithmetic has many digits of accuracy. However, string arithmetic achieves higher precision at the cost of slower calculation speed.

11.10.1 The Scale Factor

You can select a scale factor of 0 to 6. BASIC–PLUS uses the scale factor to preserve the accuracy of fractional numbers to that number of decimal places. The value 0, which is the default scale factor, is a special scale factor that disables the scaled-arithmetic feature. When the scale factor is 0, BASIC–PLUS performs calculations using standard double-precision floating-point arithmetic. (Note that your system manager can change the default scale factor to a nonzero value.)

The scale factor is set to its default value when you enter BASIC–PLUS. Each time you reenter BASIC–PLUS, the scale factor is reset to its default value. For example, you reenter BASIC–PLUS after executing system programs, such as PIP, that are not compiled with the same run-time system you are in.

BASIC–PLUS uses the scale factor when it translates a program. The scale factor in effect during translation determines how floating-point calculations will be performed when the program is executed. Suppose you specify a scale factor of 2 and then use OLD to retrieve a program. BASIC–PLUS translates the program using a scale factor of 2. Now you run the program and input a floating-point number. BASIC–PLUS internally moves the decimal point 2 places to the right and truncates it to an integer. BASIC–PLUS performs all subsequent calculations with the floating-point integers. Next, BASIC–PLUS translates the result of each arithmetic operation into a floating-point integer with the scale factor 2. On output, BASIC–PLUS moves the decimal point to the left 2 places (descales) and passes the result to the PRINT or PRINT–USING routines to format.

For example, with a scale factor of 2 in effect, the following statement causes BASIC–PLUS to move the decimal point two places to the right:

```
X = .01
```

If any rounding is necessary, it is done at this point. BASIC–PLUS converts the result, 1, to a floating-point representation. Similarly, .1 becomes 10 internally and all numbers less than .005 become 0.

The scaled arithmetic conversion thus avoids the loss of precision inherent in representing fractional numbers in binary notation. BASIC–PLUS can represent the integer accurately in floating-point format. This feature, therefore, allows more predictable arithmetic results. For example, running the following calculation without a scale factor yields a result of –.177636E–11. But running the same calculation with a scale factor yields a result of 0:

```
SCALE 2

Ready

100    X = 0.
110    X = X+.01 FOR I% = 1% TO 10000%
120    PRINT X-100
32767 END
RUNNH
 0

Ready
```

The scale factor of 2 eliminates the inaccuracy in representing a fraction two places to the right of the decimal point.

The range of integer numbers that can be accurately represented decreases according to the scale factor in effect. For example, with a scale factor of 2 in effect, two of the 15 digits represent the two digits of fraction. Thirteen places are left to accurately represent the integer portion of the number.

With a scale factor in effect, BASIC–PLUS handles output by PRINT and PRINT–USING statements in the standard manner. The PRINT statement still handles six digits or less, using the E format for numbers larger than six digits. The PRINT–USING statement formats numbers according to the specified string.

You can use the mathematical functions described in Section 9.8 in conjunction with the scaled-arithmetic feature. With a nonzero scale factor in effect, BASIC–PLUS automatically:

1. Descales the number passed
2. Computes the value of the function
3. Converts the value returned to an appropriately scaled floating-point integer

No rounding occurs; places outside the scale factor range are truncated.

11.10.2 The SCALE Command

The SCALE command lets you control the scale factor. SCALE is a BASIC–PLUS command; it cannot be used as a program statement.

BASIC–PLUS establishes the scale factor for a program during translation, not during execution. Translation occurs when you use the NEW or OLD command, and when you run a program from its source (.BAS) file. Thus, if you want to set the scale factor for a program, use the SCALE command before NEW, OLD, or RUN.

BASIC–PLUS keeps track of two scale factors: the current scale factor, used when your current program was translated, and the pending scale factor, to be used the next time translation occurs. The scale factor has a default value of 0, which means that BASIC–PLUS uses no scale factor when translating the program. On systems with the single-precision math package, the scale factor is always 0. The system manager can change the scale factor's default value on systems with the double-precision math package.

The SCALE command has two functions: it displays the two scale factors and lets you set a new pending scale factor.

The SCALE command has the format:

```
SCALE [n]
```

where n is an integer between 0 and 6.

The following examples show how the SCALE command works:

`SCALE` Displays the current scale factor and the pending scale factor. For example:

```
SCALE
 6 ,2
```

This display means that your current program was translated using a scale factor of 2; the next time translation occurs, the scale factor will be 6. One value is printed if both scale factors are the same.

`SCALE 2` Sets the scale factor to 2; this value is used the next time translation occurs. On systems with the single-precision math package, the message "?Missing special feature" is printed when you try to set the scale factor.

`SCALE 0` Disables the scaled arithmetic feature. BASIC–PLUS uses no scale factor when translating a program.

Examples

```
SCALE 2

Ready

NEW TEST

Ready

10 PRINT .01,.025,.003
20 END
RUNNH
 .01          .02          0

Ready

10 PRINT 1000.00*.075
RUNNH
 70
```

Usage Notes

1. Remember that BASIC–PLUS establishes the scale factor for a program during translation, not during execution. To change the scale factor for a program, always issue the SCALE command just before translation occurs, that is, before you use the NEW or OLD command, or before you run a program from its source (.BAS) file.
2. When you run a program, the scale factor used in computations is always the scale factor that was in effect when the program was translated.
3. If you set a new pending scale factor while you have a program in memory that was translated using a different scale factor, and then you run the program in memory, BASIC–PLUS displays the warning message "%Scale factor interlock." This message reminds you that the scale

factor you set has not yet taken effect. The program runs, but numeric results are computed using the scale factor in effect when the program was translated. (The new scale factor takes effect when you use NEW or OLD, or run a program from its source file.)

4. If you set a new pending scale factor and then run a .BAC file that was translated using a different scale factor, BASIC–PLUS again computes results using the scale factor in effect when the .BAC file was translated. In this case, BASIC–PLUS does not display the "%Scale factor interlock" message.

5. When you run a .BAC file, BASIC–PLUS changes the current scale factor to the scale factor in effect when the .BAC file was translated. The pending scale factor does not change.

   For example, suppose that the current scale factor is 2 and the pending scale factor is 6:

```
SCALE
6,2
```

   You run a .BAC file that was translated using a scale factor of 0:

```
RUN ZERO.BAC
```

   If you again display the current and pending scale factors, you see that the current scale factor is now 0 but the pending scale factor is still 6:

```
SCALE
6,0
```

6. Immediate mode statements are always translated using the current scale factor. If you set a new pending scale factor and then enter an immediate mode statement:

   - The scale factor you specify has no effect.
   - The "%Scale factor interlock" message is displayed on your terminal.

7. The scale factor is reset to its default value whenever your user job changes run-time systems. This happens when you:

   - Run a RSTS/E system program such as PIP or QUE, using RUN or a CCL command
   - Use a DCL command
   - Change your job keyboard monitor with SWITCH
   - Use another language compiler such as BASIC–PLUS–2 or COBOL

   The scale factor is also reset to its default value when you log out.

Chapter 12
Matrix Manipulation

This chapter describes the BASIC–PLUS matrix manipulation statements. These statements, called MAT statements, operate on entire matrices. MAT statements are an optional BASIC–PLUS feature, so they may not be available on your system.

Matrices (also called arrays) can be composed of variables of any type. A single matrix, however, is composed of a single type of data: floating-point, integer, or string. Note that the MAT operations in this chapter do not use the zero elements [A(0), or B(0,n) and B(n,0)] of the specified matrix.

12.1 Array Storage

You can define the size of a matrix either explicitly or implicitly. To explicitly define the size of a matrix, use the DIM statement. If you do not explicitly dimension the matrix with the DIM statement, BASIC–PLUS assumes the matrix has eleven elements in each dimension you reference. A one-dimensional matrix has eleven elements; a two-dimensional matrix has eleven rows and eleven columns. Each dimension includes a zero row and column, which are ignored by MAT statements. When you use matrices with fewer than eleven elements in each dimension, explicitly dimension them to conserve memory space.

Implicitly dimensioning the matrix A(I,J) has the same effect as explicitly including the following statement:

```
100   DIM A(10,10)
```

Dimensioning a matrix establishes two quantities for BASIC–PLUS: the maximum number of elements in each row and column, and the maximum number of elements in the matrix.

You can change the number of elements in each row and the number of columns in the matrix with the MAT statements. However, the total number of elements cannot exceed the number defined when the matrix was dimensioned. Changing the number of elements in either or both dimensions is called *redimensioning the matrix.*

When you use a matrix, be careful not to reference elements outside its dimensioned range. If the range of matrix A is 5–by–7, for example, referencing A(3,8) is improper.

Depending on which subscript is out of bounds (and by how much), either a "?Subscript out of range" error is generated or an unexpected element of the matrix will change. This result occurs because BASIC–PLUS tracks only the total size of the array, not the bounds of each subscript.

12.2 MAT READ Statement

The MAT READ statement reads the value of each element of a matrix from DATA statements. The statement has the form:

MAT READ <list of matrices>

Each element in the list of matrices indicates the maximum amount of the matrix to be read. This amount cannot be greater than the dimensioned size of the matrix. The individual elements are separated by commas. If you use the matrix name without a subscript, the entire matrix is read. For example:

```
100    DIM A%(20,20)
110    MAT READ A%
```

The preceding lines read a 20–by–20 matrix of integer data. Data is read row by row; that is, the second subscript varies most rapidly. The following statement reads a 5–by–15 matrix and redimensions the matrix A% to be 5–by–15:

```
110    MAT READ A%(5,15)
```

12.3 MAT PRINT Statement

The MAT PRINT statement prints each element of a one- or two-dimensional matrix, except elements in row 0 and column 0. The statement has the form:

MAT PRINT <matrix name> [, ;]

If the matrix name consists of an unsubscripted matrix name, BASIC–PLUS prints the entire matrix. If the matrix name is subscripted, then the subscript indicates the maximum size of the matrix to be printed. The matrix, however, is not redimensioned. Only one matrix can be output by a single MAT PRINT statement.

The semicolon and the comma determine the output format of the matrix. If the matrix name is followed by a semicolon (;), BASIC–PLUS prints out data values in a packed fashion. If a comma follows the matrix, the data values are printed across the line, one value per print zone. If neither character follows the matrix name, each element is printed on a separate line. Note the difference between matrix A; and matrix B(4,6), in the following example:

```
100     DIM A(10,10),B(20,20)
110     MAT PRINT A; &
        !PRINT 10-BY-10 MATRIX, PACKED FORMAT
120     MAT PRINT B(4,6), &
        !PRINT 4-BY-6 MATRIX, 5 ELEMENTS PER LINE
```

Line 110 prints out a 10–by–10 matrix, with no extra spaces between elements. Line 120 prints a 4–by–6 matrix with wider spacing between elements.

BASIC–PLUS can print one-dimensional arrays in either row or column format. In the following examples, V is a one-dimensional array. This statement prints the array V as a column matrix:

```
220     MAT PRINT V
```

This statement prints the array V as a row matrix, five values per line:

```
220     MAT PRINT V,
```

This statement prints the array V as a row matrix, closely packed:

```
220     MAT PRINT V;
```

For example:

```
LISTNH
100     DIM A(7), X(5)
110     MAT READ A,X
120     MAT PRINT A; &
        \ PRINT &
        \ MAT PRINT X
200     DATA 21,22,23,24,35,36,37,51,52,53,54,55
32767   END

Ready

RUNNH
 21   22   23   24   35   36   37

 51
 52
 53
 54
 55

Ready
```

Two-dimensional arrays are printed in ascending (row by row) sequence. This means that the second subscript varies most rapidly. For example:

```
LISTNH
100    DIM A(2,3)
110    FOR I% = 1% TO 2% &
       \ FOR J% = 1% TO 3% &
       \ LET A(I%,J%) = I%*100% + J% &
       \ NEXT J% &
       \ NEXT I% &
       \ PRINT
120    MAT PRINT A;
32767 END

Ready

RUNNH

 101  102  103

 201  202  203


Ready
```

12.4 MAT INPUT Statement

You can use the MAT INPUT statement to input the value of each element of a matrix. The statement has the form:

MAT INPUT <list of matrices>

BASIC–PLUS reads input from the keyboard, as with a normal INPUT statement, and prints a ? character when ready to accept the input. You can use the LINE FEED key to continue typing data on successive lines. Use the RETURN or ESCAPE key to enter the data into the system. MAT INPUT does not affect row 0 or column 0 of the matrix.

The MAT INPUT statement allows input of integer, floating-point, or string values, depending on the variable names. When more than one matrix is to be input by the same MAT INPUT statement, separate the names by commas. For example:

```
100    DIM A%(20); B(15)
110    MAT INPUT A%, B
```

These statements cause the program to input 20 integer elements for the array A% and 15 floating-point values for the array B.

Only 25 elements of the array are input in the following example, where an array or matrix element is specified:

```
200    MAT INPUT N%(25)
```

This result occurs regardless of the number of elements originally specified when the array was dimensioned. The array is then redimensioned. For example:

```
LISTNH
100    DIM A(20,20)
110    MAT INPUT A(4,3)
120    PRINT &
       \ MAT PRINT A;
32767 END

Ready

RUNNH
? 5,8,4.5,2,0,1,9.2,0,6.8,2.7,3.01,6.345

5          8          4.5

2          0          1

9.2        0          6.8

2.7        3.01       6.345

Ready
```

The matrix A is redimensioned in line 110. The INPUT statement accepts input until the entire matrix has been read or until it encounters the RETURN or ESCAPE delimiter. You can input several lines by ending the terminal line with a LINE FEED to indicate continuation on the following line.

Following the input of a matrix, the two variables NUM and NUM2 contain the number of elements input. NUM contains the number of rows input or, for a one-dimensional matrix, the number of elements entered. NUM2 contains the number of elements in the last row. The following sample program inputs a variable size matrix (up to 10–by–10):

```
LISTNH
100    DIM A(10,10)
110    INPUT 'TYPE MATRIX DIMENSIONS UP TO 10,10';N,M &
       \ MAT INPUT A(N,M)
120    PRINT 'NUM =';NUM, 'NUM2 =';NUM2
130    IF NUM*NUM2=N*M THEN PRINT 'MATRIX FILLED' &
       \ GO TO 32767
140    PRINT 'MATRIX NOT FILLED'
32767 END

Ready

RUNNH
TYPE MATRIX DIMENSIONS UP TO 10,10? 4,8
? 123,456,345,909,765,456,123,1,2,3,4,5,6,7,8,9,0,0,9,8,7,45
NUM = 3     NUM2 = 6
MATRIX NOT FILLED

Ready
```

The following statement checks the contents of the matrix:

```
MAT PRINT A;
 123   456   345   909   765   456   123   1

 2     3     4     5     6     7     8     9

 0     0     9     8     7     45    0     0

 0     0     0     0     0     0     0     0
```

Unlike the INPUT statement, the MAT INPUT statement does not permit the output of a text string. For example:

```
100    MAT INPUT 'CONTENTS OF MATRIX';A%
?Syntax error at line 100

Ready
```

12.5 Matrix Initialization Statements

A matrix initialization statement lets you create initial values for the elements of a matrix. You can also use a matrix initialization statement to redimension an array. The statement has the form:

$$\text{MAT } \langle\text{matrix name}\rangle = \left\{\begin{matrix}\text{ZER}\\ \text{CON}\\ \text{IDN}\end{matrix}\right\} [(\text{dimension(s)})]$$

where:

<matrix name>	Is the name of a predimensioned matrix.
ZER	Sets elements of the matrix to 0. (BASIC–PLUS sets all matrix elements to 0 when it creates a matrix.)
CON	Sets elements of the matrix to 1.
IDN	Sets an identity matrix; all elements are 0 except those on the diagonal, A(I,I), which are 1.

ZER, CON and IDN do not affect row 0 or column 0 of the matrix.

The optional dimensions indicate the size of the matrix. When specified, they cause the matrix to be redimensioned. When you do not specify dimensions, the existing dimensions of the matrix are assumed to be unchanged.

The following example shows the use of matrix initialization statements:

```
LISTNH
100       DIM A(10,10), B(15), C(20,20)
110       MAT A=ZER &
          !SETS ALL ELEMENTS OF A EQUAL TO 0
120       MAT B=CON(10) &
          !SETS FIRST 10 ELEMENTS OF B EQUAL TO 1
130       MAT C=IDN(10,10) &
          !SETS UP AN IDENTITY MATRIX
32767     END

Ready
```

```
RUNNH

Ready

MAT PRINT C;
 1   0   0   0   0   0   0   0   0   0
 0   1   0   0   0   0   0   0   0   0
 0   0   1   0   0   0   0   0   0   0
 0   0   0   1   0   0   0   0   0   0
 0   0   0   0   1   0   0   0   0   0
 0   0   0   0   0   1   0   0   0   0
 0   0   0   0   0   0   1   0   0   0
 0   0   0   0   0   0   0   1   0   0
 0   0   0   0   0   0   0   0   1   0
 0   0   0   0   0   0   0   0   0   1

Ready
```

12.6 Matrix Calculations

You can use matrices with mathematical operators and two intrinsic functions.

12.6.1 Matrix Operations

You can perform addition and subtraction on matrices with the common mathematical symbols. The following program uses both addition and subtraction of matrices:

```
LISTNH
100       DIM A(3), B(3), C(3)
110       A(I)=I FOR I=1, TO 3,          !ELEMENTS OF A ARE 1,2,3
120       B(I)=I*2, FOR I=1, TO 3,       !ELEMENTS OF B ARE 2,4,6
130       MAT C=A+B                      !1+2, 2+4, 3+6
140       MAT PRINT C;
150       MAT C=B-A                      !2-1, 4-2, 6-3
160       MAT PRINT C;
170       END

Ready

RUNNH
3   6   9
1   2   3

Ready
```

You can perform addition and subtraction only on matrices that are the same size. The following example shows that you cannot indicate a subset of one matrix as part of an operation:

```
110    DIM A(50), B(25), C(50)
120    MAT C=A+B

RUNNH
? Matrix dimension error at line 120
```

For line 120 to execute properly, line 110 should read:

```
110    DIM A(50), B(50), C(50)
```

There are two kinds of multiplication for matrices: scalar multiplication and multiplication of conforming matrices. You can multiply conforming matrices as follows:

```
100    DIM D(10,5), C(5,10), R(10,10)
110    MAT R=D*C
```

D, C, and R in line 110 are *conforming matrices* because the number of columns in matrix D equals the number of rows in matrix C. The dimensions of the matrix R must be large enough to contain the number of columns in D and the number of rows in C.

You can perform scalar multiplication of a matrix as follows:

```
150    MAT C = (K)*A
```

Each element of matrix A is multiplied by the scalar value (constant, variable, or expression) K, indicated in parentheses. The form MAT A=(K)*A is also legal.

Matrix A can be copied into matrix C (providing enough space is available in matrix C) as follows:

```
160    MAT C = A
```

12.6.2 Matrix Functions

Three functions allow transposition and inversion of matrices:

TRN – Transposes the matrix
INV – Inverts the matrix
DET – Returns the value of the determinant of the matrix after inversion

The following statement causes matrix C to be set equal to the transposition of matrix A:

```
150    MAT C = TRN(A)
```

That is, C(I,J)=A(J,I) for all I,J; matrix C is redimensioned if necessary. For example:

```
10    DIM X(15,25), N(5,10), M(5,5)
90    MAT N=INV(M)
100   PRINT "DETERMINANT OF MATRIX M IS"; DET
```

This causes N to be computed as the inverse of matrix M (M must be a square matrix). After the inversion is complete, the function DET is set to the value of the determinant of matrix M. (If the matrix being inverted is singular, and thus cannot be inverted, the message "?Can't invert matrix" is printed.) You can use the value of DET in subsequent computations, as you use any other function. For example:

```
LISTNH
200    MAT A = INV(X) &
       \D1=DET
210    MAT B = INV(A) &
       \D2=DET
220    IF D1=1/D2 GO TO 340 ELSE PRINT 'RELATIONSHIP TRUE'

Ready
```

Matrix inversion, like the other BASIC–PLUS matrix operations, does not operate on the elements of the row 0 and column 0 of the matrix. Inversion, however, destroys the previous contents of these elements. The operation MAT A = INV(A) is legal.

Chapter 13
Advanced Statements and Features

This chapter describes advanced BASIC–PLUS statements and features. Several statements in this chapter are more advanced forms of statements described in Chapter 9.

This chapter describes the following topics:

Topic	Statements or Functions
Multiple Line Function Definitions	DEF*
Conditional Transfer of Control	ON–GOTO, ON–GOSUB, IF–THEN–ELSE
Conditional Termination of FOR Loops	FOR–WHILE, FOR–UNTIL
Statement Modifiers	IF, UNLESS, FOR, WHILE, UNTIL
Error Handling	ON ERROR GOTO, RESUME
System Functions	DATE$, TIME$, TIME
Suspending Program Execution	SLEEP, WAIT
Transferring Control Between Programs	CHAIN

13.1 DEF* Statement, Multiple-Line Function Definitions

The DEF* statement, described in Section 9.9, shows how to create a one-line function that you can use as an element in a BASIC–PLUS statement. You can also use DEF* to define multiple-line functions. The format for a multiple-line function definition is:

```
DEF* FN<variable(arguments)>
<body of definition>
FNEND
```

As in the single-line DEF* function, the arguments can be zero to five dummy variables of any type or mixture of types. However, unlike the single-line DEF* function, the multiple-line DEF* function has no equal sign after the function name on the first line.

Within the multiple-line definition, you should have a statement of the form:

[LET] FN<variable> = <expression>

The DEF* function returns the value of FN<variable> when it reaches the FNEND statement.

The value of this expression is returned as the value of the function. There can be more than one such statement. The following example returns the larger of the two numbers:

```
100   DEF* FNM(X,Y)
110   LET FNM=X
120   IF X<=Y THEN GOTO 140
130   LET FNM=Y
140   FNEND
```

The next example shows a recursive function that computes N-factorial. The term "recursive" refers to the repetitive process where the result of each cycle depends on the result of the previous cycle.

```
LISTNH
100       DEF* FNF(M%)
105       IF M% = 0% THEN FNF = 1 &
          \ GO TO 120
110       IF M% = 1% THEN FNF = 1 &
                    ELSE FNF = M% * FNF(M%-1%)
120       FNEND
130       INPUT 'VALUE FOR FACTORIAL'; M%
140       PRINT M%; 'FACTORIAL EQUALS'; FNF(M%)
32767     END

Ready

RUNNH
VALUE FOR FACTORIAL? 6
 6 FACTORIAL EQUALS 720

Ready

RUNNH
VALUE FOR FACTORIAL? 0
 0 FACTORIAL EQUALS 1

Ready
```

Any non-dummy variable you reference in a function definition has the current value of that variable in your program. These variables retain any value you assign during execution. Also note that not all context information is saved in functions. Function parameters are saved and restored, but variables and information stored in temporary locations are not saved.

You can nest multiple-line DEF* function calls; that is, one multiple-line function can reference another. A multiple-line function lets you transfer control outside its boundaries with GOSUB, ON ERROR... RESUME, nested function references, or GOTO. However, you must follow certain rules when you transfer control out of a function or enter a function from outside its boundaries. For example, once a function is entered, it must be exited through its FNEND statement. Also note that an error occurs if BASIC–PLUS encounters an FNEND statement whose companion DEF* FN... statement has not been executed. You should avoid transfers out of function boundaries—they may not execute as expected in other versions of BASIC.

The parameters you use to call a user-defined function are formal. This means that their former values are retained after the function has been invoked. For example:

```
LISTNH
100    DEF* FNB (X)
110    X=0 &
       \FNB=10
120    FNEND
200    A=1 &
       \B=FNB(A) &
       \PRINT A,B
32767  END

Ready

RUNNH
 1         10

Ready
```

Note that A is not set to 0 by the function FNB(A).

You can write functions using a variety of argument types. For example:

```
LISTNH
100    DEF* FNA$(A,B,C%)
110    IF A>B THEN FNA$ = CHR$(A+C%) &
              ELSE FNA$ = CHR$(A+1)
120    FNEND
200    INPUT 'VALUES FOR A,B,C%';A,B,C%
210    PRINT 'FNA$(A,B,C%) = ';FNA$(A,B,C%)
32767  END

Ready

RUNNH
VALUES FOR A,B,C%? 36,7,5,24
FNA$(A,B,C%) = <

Ready

RUNNH
VALUES FOR A,B,C%? 45,2,5,67,8
FNA$(A,B,C%) = 5

Ready
```

13.2 ON–GOTO Statement

The simple GOTO statement transfers control of the program to another line number. The ON–GOTO statement transfers control to one of several lines, depending on the value of an expression, when the statement is executed. The statement has the form:

ON <expression> GOTO <list of line numbers>

BASIC–PLUS evaluates the expression, using the integer part as an index to one of the line numbers in the list. For example:

```
50 ON X GOTO 100,200,300
```

This statement transfers control to line number:

100 if X is greater than or equal to 1. and less than 2.

200 if X is greater than or equal to 2. and less than 3.

300 if X is greater than or equal to 3. and less than 4.

Note that the preceding line 50 is equivalent to:

```
50 IF X=1 THEN &
     GOTO 100 &
   ELSE IF X=2 THEN &
     GOTO 200 &
     ELSE IF X=3 THEN &
       GOTO 300
```

However, the ON–GOTO statement in line 50 requires less storage space and executes faster.

Values of X out of the range 1, 2, 3 in this example cause the error message:

```
?On statement out of range at line 50
```

However, you can transfer to an ON ERROR GOTO routine, checking for ERR=58. (See Section 13.7.)

13.3 ON–GOSUB Statement

The GOSUB statement transfers control of your program to a subroutine; the RETURN statement returns control from that subroutine to normal program execution (see Section 9.10 for details). The ON–GOSUB statement conditionally transfers control to one of several subroutines or to one of several entry points in one (or more) subroutine(s). (See ON–GOTO conditions, Section 13.2.) The statement has the form:

ON <expression> GOSUB <list of line numbers>

The integer value of the expression determines which line number control transfers to. When the RETURN statement is executed, control transfers to the line following the ON–GOSUB line.

Consider the following example:

```
80 ON X%-Y% GOSUB 900,933,1014
```

When line 80 is executed, the value of X%–Y% is calculated. If the value is 1, control transfers to line 900; if the value is 2, control goes to line 933; if the value is 3, control goes to line 1014. If the quantity X%–Y% is not equal to 1, 2 or 3, the following error message is printed:

```
?On statement out of range at line 80
```

You can transfer to an ON ERROR GOTO routine with ERR=58 (see Section 13.7).

Since you can transfer into a subroutine at different points, you can use the ON–GOSUB statement to determine which portion of the subroutine should execute.

13.4 IF–THEN–ELSE Statement

The IF–THEN statement transfers control to another line or executes a specified statement, depending on a stated condition.

The IF–THEN–ELSE statement is the same as the IF–THEN statement. But instead of executing the line that follows the IF statement, you can specify another line number or statement for execution when the condition is not met. The statement has the form:

IF <condition> { THEN <statement> / THEN <line number> / GOTO <line number> } [ELSE <statement> / ELSE <line number>]

The <condition> is either a relational expression or a logical expression.

A relational expression is defined as:

<expression> <relational operator> <expression>

The relational expression is described in Section 8.4.3.

A logical expression is one of the following:

1. An integer expression (FALSE if 0, TRUE if <>0)
2. A set of relational expressions, connected by logical operators
3. A set of integer expressions, logical expressions, or both, connected by logical operators

The condition is tested; if it is true, the system executes the THEN or GOTO part of the statement. If the condition is false, the ELSE part of the statement is executed.

Here is an example of an IF–THEN–ELSE statement:

```
75 IF X>Y THEN PRINT "GREATER" ELSE PRINT "NOT GREATER"
```

Another IF statement can follow either the THEN or ELSE clause in the preceding example. In this way, you can nest the IF statement to any level you want. Use line continuation and spaces or tabs to make these complex statements easier to read. For example:

```
LISTNH
100 ·  INPUT 'ENTER THREE NUMBERS';A,B,C
110    IF A>B THEN &
               IF B>C THEN PRINT 'A>B>C' &
                       ELSE IF C>A &
                               THEN PRINT 'C>A>B' &
                               ELSE PRINT 'A>C>B' &
               ELSE IF A>C THEN PRINT 'B>A>C' &
                       ELSE IF B>C &
                               THEN PRINT 'B>C>A' &
                               ELSE PRINT 'C>B>A'
32767 END

Ready

RUNNH
ENTER THREE NUMBERS? 4.6,-.01,-3.5
A>B>C

Ready

RUNNH
ENTER THREE NUMBERS? 2005,2.8,3006
C>A>B
```

The IF–THEN–ELSE statement can appear anywhere in a multi-statement line. However, if any other statements follow, all statements up to the next line number are part of the THEN or ELSE clause. For example:

```
210 IF A=1.0 THEN GOTO 100 ELSE PRINT A\ PRINT 'ABNORMAL'
```

The value of A and the text string ABNORMAL are both printed when A<>1.

```
20 IF A>B THEN IF B<C THEN PRINT "B<C" &
   \ GOTO 30
25 PRINT"A<B"
30 STOP
```

The statement GOTO 30 is encountered and executed only when "B<C" is printed. If either A<B or B>C, the line "A<B" is printed.

13.5 Conditional Termination of FOR Loops

In the simple FOR–NEXT loop described in Section 9.5.1, the format of the FOR statement is given as:

FOR <variable> = <expression> TO <expression> [STEP <expression>]

Often you do not know the final value of the loop variable, and you want to execute the loop many times to satisfy some condition. This condition might be the point where further iterations of a function contribute no accuracy to the result. BASIC–PLUS provides a convenient way of specifying that a loop is executed until a condition is detected or while some condition is true. These statements take the form:

FOR <variable> = <expression> [STEP <expression>] { WHILE / UNTIL } <condition>

The <condition> is either a relational expression or a logical expression. BASIC–PLUS evaluates the condition before the loop executes and also at each loop iteration. The iteration proceeds if the result is true (FOR–WHILE) or false (FOR–UNTIL).

There is a notable difference between a FOR loop with WHILE or UNTIL and one with a terminal value for the loop variable. Consider the two loops in the following program:

```
LISTNH
10     FOR I%=1% TO 10% &
       \ PRINT I%; &
       \ NEXT I%
20     PRINT 'I%=';I%
50     FOR I%=1% UNTIL I%>10% &
       \ PRINT I%; &
       \ NEXT I%
60     PRINT 'I%=';I%
32767  END

Ready

RUNNH
 1  2  3  4  5  6  7  8  9  10  I%=  10
 1  2  3  4  5  6  7  8  9  10  I%=  11

Ready
```

Each loop prints the numbers from 1 to 10. When the loop at line 10 is done, however, the loop variable is set to the last value used (that is, 10). In the second loop beginning at line 50, the loop variable is set to the value that caused the loop to be terminated (that is, 11).

Now consider the next two loops:

```
LISTNH
10      X=10
20      FOR I=1 TO X &
        \ X=X/2 &
        \ PRINT I,X &
        \ NEXT I
30      PRINT &
        \ X=10
40      FOR I=1 UNTIL I>X &
        \ X=X/2 &
        \ PRINT I,X &
        \ NEXT I
32767 END

Ready

RUNNH
 1            5
 2            2.5
 3            1.25
 4            .625
 5            .3125
 6            .15625
 7            .078125
 8            .390625E-1
 9            .195313E-1
 10           .976563E-2

 1            5
 2            2.5

Ready
```

In the loop beginning with line 20, the iteration stops when I exceeds the initial value of X (that is, 10). Even though the value of X changes in the loop, the initial value of X determines the performance of the loop.

In the second loop, the current value of X determines when the iteration ceases. Thus, after three iterations, I is greater than X in the second loop and the program ends. (When you omit the STEP value, it is assumed to be 1.)

These forms of loop control are useful in iterative applications where data generated during the loop execution determines loop completion.

Consider the problem of scanning a table of values until two successive elements are both 0 or the end of the table is reached:

```
100    FOR I%=1% UNTIL I%=N% OR X(I%)=0 AND X(I%+1%)=0
200    PRINT X(I%)
300    NEXT I%
```

The following two programs also illustrate the FOR–UNTIL and FOR–WHILE constructs:

```
LISTNH
100    INPUT 'LETTER IS';Y$
110    X$=''
120    FOR I%=0% UNTIL X$=Y$ OR X$ = 'ZZZ' &
       \ READ X$ &
       \ NEXT I%
130    IF X$ = 'ZZZ' THEN PRINT 'IMPROPER INPUT' &
       \ GO TO 32767
140    PRINT 'LETTER IS NUMBER';I%; 'IN ALPHABET'
500    DATA A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,ZZZ
32767 END

Ready

RUNNH
LETTER IS? J
LETTER IS NUMBER 10 IN ALPHABET

Ready

RUNNH
LETTER IS? 9
IMPROPER INPUT

Ready

RUNNH
LETTER IS? X
LETTER IS NUMBER 24 IN ALPHABET

Ready

RUNNH
LETTER IS? CC
IMPROPER INPUT

Ready

LISTNH
100    INPUT 'WORD';Y$
110    X$=''
120    FOR I% =0% WHILE X$<=Y$ &
       \ READ X$ &
       \ NEXT I%
130    IF I% = 1% THEN PRINT 'IMPROPER INPUT' &
       \ GO TO 32767
140    PRINT 'WORD BEGINS WITH LETTER';I%-1%
500    DATA A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,ZZZ
32767 END

Ready

RUNNH
WORD? SECOND
WORD BEGINS WITH LETTER 19

Ready

RUNNH
WORD? /MESSAGE
IMPROPER INPUT

Ready
```

13.6 Statement Modifiers

BASIC–PLUS provides five statement modifiers to increase the flexibility and ease of expression in a program line: IF, UNLESS, FOR, WHILE, and UNTIL. Append these modifiers to program statements to conditionally execute statements or create implied FOR loops. All statement modifiers operate on only one statement.

Where possible, use statement modifiers instead of IF statements and loops. Your programs will be more efficient.

13.6.1 IF Statement Modifier

The IF statement modifier has the form:

<statement> IF <condition>

It is analogous to the form IF <condition> THEN <statement>.

For example:

```
10  PRINT X IF X<>0
```

This statement is the same as:

```
10  IF X<>0 THEN PRINT X
```

BASIC–PLUS executes this statement if the condition is true.

Because the IF statement modifier affects only one statement, you can use fewer line numbers in a program when you use IF statement modifiers instead of IF statements. Each line number in a program uses memory space; thus, using fewer line numbers makes a program more efficient. For example, suppose you want to print two values, X and A. You want to print X only if a specified condition is true, but you always want to print A. To write this code using an IF statement, you need two line numbers:

```
10    IF X<>0 THEN PRINT X
20    PRINT A
```

If you write these statements on a multi-statement line, A is printed only when X<>0 is true. Otherwise, control goes to the next line in the program:

```
10    IF X<>0 THEN PRINT X &
      \ PRINT A
20    END
```

On the other hand, you can write this code using one line number when you use the IF statement modifier:

```
10    PRINT X IF X<>0 &
      \ PRINT A
```

When a statement modifier appears to the right of an IF–THEN statement, the modifier operates on either the THEN clause or the ELSE clause, depending on the modifier's placement to the left or right of ELSE. For example:

```
100   IF 1=1 THEN PRINT "HELLO" ELSE PRINT "BYE" IF 2=2
```

This statement prints HELLO because the test 1=1 is true. The modifier IF 2=2 is also true, but it applies only to the ELSE clause and is never tested. You cannot include an ELSE clause when using IF as a modifier.

You can use several modifiers within the same statement. For example:

```
70   PRINT X(I,J) IF I=J IF X(I,J)<>0
```

Line 70 prints the value of X(I,J) only if the value of X(I,J) is not zero and if I equals J. When more than one modifier is on a line, the modifiers are executed in a right-to-left order. The term *nested modifiers* describes this situation.

13.6.2 UNLESS Statement Modifier

The UNLESS statement modifier causes a statement to be executed if a condition is false. It has the form:

<statement> UNLESS <condition>

This form simplifies the negation of a logical condition. For example, the following statements are equal:

```
10 PRINT A UNLESS A=0
10 PRINT A IF NOT A=0
10 IF NOT A=0 THEN PRINT A
10 IF A<>0 THEN PRINT A
```

13.6.3 FOR Statement Modifier

The FOR statement modifier creates an implied FOR loop on one line. This modifier has the form:

<statement> FOR <variable> = <exp> TO <exp> [STEP <exp>]

$$\left[\left\{\begin{matrix}\text{WHILE}\\ \text{UNTIL}\end{matrix}\right\}\ \text{<condition>}\right]$$

An example that uses none of the optional elements is:

```
10 PRINT I, SQR(I) FOR I=1, TO 10,
```

This statement is equivalent to the following FOR–NEXT loop:

```
20    FOR I=1 TO 10 &
      \PRINT I,SQR(I) &
      \NEXT I
```

NOTE

An implied FOR loop executes only one statement in the program.

Like all modifiers, a FOR modifier in an IF statement operates only on the associated THEN or ELSE clause. It never operates on the conditional expression to the left of THEN. Thus, if you intend to print nonzero values in a matrix X(100), the following program does not operate properly:

```
10    DIM X(100%)
15    READ X(I%) FOR I%=1% TO 100%
20    IF X(I%)<>0, THEN PRINT I%;X(I%); FOR I%=1% TO 100%
```

The implied FOR loop at line 20 applies only to the THEN PRINT... part of the statement and not to the IF... part. The first value of X that the system tests is X(100), since I remained at 100 from statement 15.

For proper operation, line 20 should be a PRINT statement with nested modifiers:

```
20    PRINT I%;X(I%), IF X(I%)<>0, FOR I%=1% TO 100%
```

In this form, the nested modifier rule tests and prints the values of X(I%) appropriately.

13.6.4 WHILE Statement Modifier

The WHILE statement modifier repeatedly executes a statement while a specified condition is true. It has the form:

<statement> WHILE <condition>

For example:

```
10      LET X=X^2% WHILE X^2%<16000
```

The preceding line 10 is a more efficient way of writing the loop than the following WHILE–NEXT construct:

```
10      WHILE X^2%<16000
20      LET X=X^2%
30      NEXT
```

The WHILE statement is also more efficient than the following IF–THEN construct:

```
10      LET X=X^2% &
        IF X^2<16000 THEN 10
```

The WHILE modifier (and the UNTIL modifier described in Section 13.7.5) is useful only in iterative loops, where the logical loop structure modifies the values that terminate the loop. This differs significantly from FOR loops, where the control variable automatically iterates. A WHILE statement does not automatically increment a control variable. The following infinite loops never terminate:

```
10    X=X+1, WHILE I<1000
15    PRINT I,A(I) WHILE A(I)<>0
```

In both cases, the program fails to change the values that terminate the loop.

A correct use of the WHILE modifier is:

```
110    X=X+1, WHILE X<100,
120    PRINT X
32767 END

Ready

RUNNH
 100

Ready
```

13.6.5 UNTIL Statement Modifier

The UNTIL statement modifier repeatedly executes a statement until a condition is true. It has the form:

<statement> UNTIL <condition>

For example:

```
10 X=X+1, UNTIL X<>SQR(X^2%)
20 PRINT X
```

The preceding example is more efficient than the UNTIL–NEXT construct in this example:

```
100      UNTIL X<>SQR(X^2%)
200      X=X+1,
300      NEXT
400      PRINT X
```

The UNTIL statement is also more efficient for this loop than the IF–THEN–ELSE construct:

```
100       IF X = SQR(X^2%) THEN GO TO 110 ELSE GO TO 130
110       X = X+1.
120       GO TO 100
130       PRINT X
```

13.6.6 Multiple Statement Modifiers

You can use more than one modifier in a single statement. BASIC–PLUS processes multiple modifiers from right to left. For example:

```
100    LET A=B IF A>0 IF B>0
```

Line 100 is equivalent to each of the following:

```
100    IF B>0 THEN IF A>0 THEN A=B

100    IF B>0 AND A>0 THEN LET A=B

100    IF B<=0 THEN 150
110    IF A<=0 THEN 150
120    LET A=B
150    ! TEST OF A AND B COMPLETE
```

The following statement causes BASIC–PLUS to read a two-dimensional matrix (M by N) one row at a time:

```
150  READ A(I,J) FOR J=1 TO M FOR I=1 TO N
```

Each of the following examples is equivalent to the preceding line:

```
150    MAT READ A(N,M)
```

```
150    FOR I=1 TO N
160    FOR J=1 TO M
170    READ A(I,J)
180    NEXT J
190    NEXT I
```

Section 13.6.3 describes the interaction of FOR and IF modifiers.

13.7 Error Handling

BASIC–PLUS detects certain errors during program execution. These errors fall into two broad areas:

1. Computational errors (such as division by 0)
2. I/O errors (such as reading an end-of-file character (CTRL/Z) as input to an INPUT statement)

BASIC–PLUS normally does two things when it detects an error: it prints an error message and terminates your program. However, BASIC–PLUS allows you to continue program execution after it encounters some errors. In these cases you can use the ON ERROR GOTO statement to tell BASIC–PLUS that a subroutine exists at a specified line number to analyze errors and attempt to recover from them. This section describes ON ERROR GOTO and two other features used in error handling, the RESUME statement and the ERL variable.

13.7.1 ON ERROR GOTO Statement

The format of the ON ERROR GOTO statement is:

ON ERROR GOTO [<line number>]

Place this statement before any executable statements that will use the error handling routine. If an error occurs, the system interrupts execution of your program and transfers control to your error subroutine at the line number indicated. The variable ERR assumes one of the values listed in Appendix C.

When BASIC–PLUS encounters an error in your program, it checks to see if the program has executed the ON ERROR GOTO statement. If not, a message prints at your terminal and the program proceeds (that is, if the error does not terminate execution). If the ON ERROR GOTO statement was executed, the program continues at the specified line number. The program can test the variable ERR to discover what error occurred and to decide what action to take.

NOTE

An ON ERROR GOTO statement with an incorrect target statement number can cause error messages that are confusing or inappropriate.

13.7.2 RESUME Statement

The RESUME statement in an error handling routine is like a RETURN statement in a subroutine. After the error handling routine has processed the error, the RESUME statement allows your program to exit from the ON ERROR GOTO routine and continue execution. The RESUME statement causes execution to be continued at the program line that caused the error. Place the RESUME statement at the end of the error handling routine. The format of the RESUME statement is:

RESUME [<line number>]

For example:

```
2000 RESUME
```

Line 2000 restarts your program at the line where BASIC–PLUS detected the error. To continue execution at some other point (as for a noncorrectable problem), specify the new line number at the end of the error handling routine. For example:

```
2001 RESUME 100
```

When a RESUME or RESUME 0 statement passes control to a multistatement line, control goes to the DIM, DEF*, FNEND, FOR, NEXT, or DATA statement immediately preceding the statement that caused the error. If none of these six statements are on the line, BASIC–PLUS passes control to the first statement on the line. The first statement on a multistatement line should be the one that is most likely to generate a trappable error.

Consider the line:

```
50 A=A+1 \ PRINT A \ FOR M%=1% TO 3% &
         \ INPUT X(M%) &
         \ NEXT M%
```

If there is an error in the INPUT statement, control passes to the preceding FOR statement on the same line, instead of to the first statement on the line. But BASIC–PLUS does not reinitialize the loop; it retries the INPUT statement without changing M%.

If an error handling routine contains code that can generate errors, it should execute a RESUME statement before continuing to process the original error.

13.7.3 Disabling the Error Handling Subroutine

In certain portions of a program, you may want BASIC–PLUS, and not your program, to process errors. You can disable the error handling subroutine by executing one of the following statements:

```
100    ON ERROR GOTO 0
100    ON ERROR GOTO
```

These statements return control of error handling to the system. In the second form, line 0 is assumed. Note that you should specify 0 for compatibility with BASIC–PLUS–2. When you execute either statement, BASIC–PLUS treats errors as if no ON ERROR GOTO had executed.

Generally, your error handling subroutine will detect and handle only a few specific errors. It is useful to have BASIC–PLUS handle other errors as they occur. For this reason, your program can execute the ON ERROR GOTO statement inside the error subroutine.

When the ON ERROR GOTO 0 statement is executed in an error-handling routine, disabling occurs retroactively. The system reports the error that caused the error subroutine to execute. A message is printed as though the ON ERROR GOTO statement had not been in effect.

The following example shows how an error handling routine can help inexperienced users interact with a BASIC–PLUS program. These users may not know what to type at the terminal, so the program prompts them. The program uses the WAIT statement to allow up to 60 seconds for the user to respond (see Section 13.9). After 60 seconds, the error "?Keyboard WAIT exhausted" signals the program that the user has not replied. Then the program prints additional information for the user.

The program requests your name with the INPUT statement on line 120; line 110 allows you 60 seconds to respond. The system executes the ON ERROR GOTO statement on line 100.

```
LISTNH
100    ON ERROR GOTO 1000 &
       ! SET UP ERROR ROUTINE
110    WAIT 60 &
       ! WAIT 60 SECONDS FOR REPLY
120    INPUT 'YOUR NAME' ;N$ &
       ! GET STUDENT NAME
150    STOP

1000   ! THIS IS THE ERROR HANDLING ROUTINE
1010   IF ERR<>15 THEN ON ERROR GO TO 0 &
       ! WAIT ERRORS ONLY
1020   PRINT &
       ! SKIP TO NEW LINE
1030   PRINT 'PLEASE TYPE YOUR NAME' &
       \ PRINT 'AND THEN HIT "RETURN" KEY'
1050   RESUME &
       ! TRY AGAIN
32767  END
```

If some error other than "?Keyboard WAIT exhausted" (ERR=15) called the error subroutine, the program exits through the ON ERROR GOTO 0 in line 1010. This permits the system to print the appropriate error message at your terminal. Note that exiting through the RESUME at line 1050 causes the INPUT statement to be executed again.

13.7.4 The ERL Variable

Sometimes you need to know the line number where an error occurred. After error detection, the integer variable ERL contains the line number of the error. (The exception is the "?Programmable ^C trap" error, (ERR=28); see the *RSTS/E Programming Manual*. In this case ERL is not set, but the LINE variable is set to the line number executing when CTRL/C was typed.)

For example, you can use ERL to indicate which INPUT statement caused an "?End of file on device" error. But you must be careful when using the

ERL variable. When you change or resequence the line numbers of statements within the program, you can alter the value of the ERL variable within an expression context. For example:

```
100    ON ERROR GOTO 1000
110    INPUT ' TYPE TWO NON-ZERO NUMBERS'; A,B
120    LET X=A/B
130    LET X=X+B/A
140    PRINT X
150    STOP
       .
       .
       .
1000   IF ERR<>61 THEN ON ERROR GOTO 0
1010   PRINT 'FIRST NUMBER WAS 0' IF ERL=130
1020   PRINT 'SECOND NUMBER WAS 0' IF ERL=120
1030   RESUME 32767
32767  END

Ready

RUNNH
TYPE TWO NON-ZERO NUMBERS? 5,10
 2.5
Stop at line 150

Ready

RUNNH
TYPE TWO NON-ZERO NUMBERS? 6,0
SECOND NUMBER WAS 0

Ready

RUNNH
TYPE TWO NON-ZERO NUMBERS? 0,7
FIRST NUMBER WAS 0

Ready
```

If you move the LET statements in lines 120 and 130 to some other line numbers, lines 1010 and 1020 also require a change.

13.8 System Functions

BASIC–PLUS has several system functions that get information about or perform operations with the system. Table 13–1 describes these functions.

Table 13–1: System Functions

Function	Meaning	Sample Usage
DATE$(0%)	Returns the current day, month and year. Note that the date contains both upper- and lowercase letters. If your system uses the numeric date format, DATE$ returns the date as year, month, and day.	PRINT DATE$(0) 10-Jan-79 Ready 76,03,22

(continued on next page)

Table 13–1: System Functions (Cont.)

Function	Meaning	Sample Usage
DATE$(N%)	Returns a character string corresponding to a calendar date. The formula used to translate between N and the date is: (day of year)+[(number of years since 1970)*1000] DATE$(1%) = "01–Jan–70" DATE$(2060%) = "29–Feb–72"	
	If your system uses the numeric date format, DATE$ returns the date as year, month, and day.	`76.02.29`
TIME$(0%)	Returns the current time of day as a character string.	`75 IF TIME$(0)>="05:45 PM"` `  THEN PRINT "TIME TO QUIT"`

TIME$(N%)	Returns a string corresponding to the time at N% minutes before midnight.
	
	TIME$(1%) = "11:59 PM"
	or "23:59 "
	TIME$(1440%) = "12:00 PM"
	or "00:00 "
	TIME$(721%) = "11:59 AM"
	or "11:59 "
	
	N% must be less than 1441 to return a valid string. The system manager determines whether your system uses the AM/PM or 24–hour time format (for example, 02:40 PM or 14:40).
		`PRINT TIME$(1%)`
		`11:59 PM`
		
		`Ready`
		
		`PRINT TIME$(1400%)`
		`12:40 AM`

TIME(0%)	Returns the clock time in seconds since midnight.	`25 IF TIME(0)>43200` `  THEN PRINT "AFTERNOON"`
TIME(1%)	Returns the central processor (CPU) time used for the job in 0.1 second quanta.	`10 IF TIME(1)>30 THEN STOP`
TIME(2%)	Returns the *connect time* (duration of time that you have been logged onto the system) for the job in minutes.	`10 IF TIME(2)>1000 THEN STOP`
TIME(3%)	Returns the number of kilo-core ticks (KCTs) that your job used. (See the *RSTS/E System User's Guide* for an explanation of KCTs.)	`80 PRINT TIME(3)`
TIME(4%)	Returns the device time for the job in minutes. The time is accumulated by the RSTS/E monitor.	`40 IF TIME(4)/60>2.5 THEN 90`

The string that the TIME$ function returns conforms to standard usage. The string is always 8 characters in length. For 24–hour time, the format is "hh:mm ", where midnight is "00:00 " and noon is "12:00 ". For AM/PM time, the format is "hh:mm xx", where midnight is "12:00 PM" and noon is "12:00 M ". Table 13–2 shows examples of these formats.

Table 13–2: TIME$ String Examples

24–hour Format	AM/PM Format	Description
00:00	12:00 PM	Midnight
00:01	12:01 AM	1 minute after midnight
00:59	12:59 AM	59 minutes after midnight
01:00	01:00 AM	1 hour after midnight
11:59	11:59 AM	1 minute before noon
12:00	12:00 M	Noon
12:01	12:01 PM	1 minute after noon
12:59	12:59 PM	59 minutes after noon
13:00	01:00 PM	1 hour after noon
23:59	11:59 PM	1 minute before midnight

In addition to the functions listed in Table 13–1, BASIC–PLUS has three other functions that allow you to perform more complex operations with the RSTS/E system:

SYS	Lets you communicate with the RSTS/E monitor.
SPEC%	Performs special operations on disks, flexible diskettes, magnetic tape, terminals, and pseudo keyboards.
MAGTAPE	Performs special operations on magnetic tape.

These functions are described in the *RSTS/E Programming Manual.*

13.9 SLEEP and WAIT Statements

You can use two special statements in a BASIC–PLUS program: SLEEP and WAIT. Both statements let you suspend your program for a stated interval.

The SLEEP statement has the form:

```
SLEEP <expression>
```

SLEEP suspends the running program for the number of seconds that the expression indicates. After this period the program runs again. Thus, you request these seconds of idle time. To call a job from SLEEP before the specified number of seconds expires, type a delimiter (RETURN, LINE FEED, FORM FEED, or ESCAPE) at your job terminal. You can also call a job from SLEEP at any other terminal that is opened by your job.

The following program segment overrides line terminating delimiters and insures continuous SLEEP for a specified time:

```
100    T=TIME(0)
110    SLEEP T+30-TIME(0) &
       \ IF TIME(0)-T<30 GOTO 110
120    INPUT X
```

In this program, the INPUT statement is executed only if the time elapsed is equal to or greater than 30 seconds. But if you type a delimiter, SLEEP is executed again for the length of time remaining in the original 30 seconds or until you type another line terminating character.

A job is also awakened when the system manager disables logins or one of several other conditions occurs. See the *RSTS/E Programming Manual* for details.

The WAIT statement has the form:

WAIT <expression>

WAIT sets a maximum period for the system to wait for input from your keyboard. If you do not type a delimiter (RETURN, LINE FEED, ESCAPE, or FORM FEED) within the number of seconds that the expression specifies, program execution continues and a "?Keyboard WAIT exhausted" error (ERR = 15) occurs. You can detect this error using ON ERROR GOTO.

The WAIT statement works with the INPUT statement. For example:

```
LISTNH
10    ON ERROR GOTO 100
20    WAIT 15
30    INPUT '16 + 16 =';A
40    WAIT 0
50    IF A=32 THEN PRINT 'RIGHT!' &
                ELSE PRINT 'NO, TRY AGAIN' &
                \ GOTO 10
60    STOP
100   IF ERR<>15 THEN ON ERROR GOTO 0
110   PRINT 'WAKE UP!'
120   RESUME 30
32767         END

Ready

RUNNH
16 + 16 =? WAKE UP!
16 + 16 =? 30
NO, TRY AGAIN
16 + 16 =? 32
RIGHT!
Stop at line 60

Ready
```

In this example, the WAIT statement at line 20 causes the system to wait 15 seconds for keyboard input when the INPUT statement is executed. Line 100 executes only if you fail to respond within 15 seconds.

A WAIT statement affects the entire program. The specified wait period is in effect for all terminal input until another WAIT statement is executed. Use the WAIT 0 statement to restore the terminal to its normal state where no timeout occurs. In this state, the system waits until a line is entered, however long it takes.

13.10 CHAIN Statement

The CHAIN statement transfers control from one BASIC–PLUS program to another. Use CHAIN when a program is too large to load into memory and run in one operation. Break the program into two or more separate programs and then use the CHAIN statement in these separate programs to call other programs into memory to be run.

NOTE

This discussion of the CHAIN statement uses terms that may be unfamiliar to you. These terms relate to working with data files. Part IV of this manual discusses data files in detail.

The CHAIN statement has the format:

CHAIN <string> [[LINE] <line number>]

The string is a file specification that names the program for BASIC–PLUS to load, translate, and execute. Only the file name is required; all other parts of the file specification are optional. LINE, which means line number, is an optional word included for compatibility with BASIC–PLUS–2. The line number, if included, specifies the line where execution is to start. If you omit the line number, execution starts at the first line in the program. Consider the following examples:

```
1000 CHAIN "MAIN.BAC"

1000 CHAIN "MAIN.BAC" LINE 2000

1000 CHAIN "MAIN.BAC" 2000
```

All three statements load and execute the program MAIN.BAC. In the first statement, execution starts at the first line in MAIN.BAC; in the second and third statements, execution starts at line 2000.

The CHAIN statement works much like the RUN command. Like RUN, it loads and executes a translated (.BAC) file by default. If no translated program exists, it loads, translates, and executes the corresponding source (.BAS) file. Because .BAS files require translation each time they are run, it is more efficient to chain to .BAC files. The system displays an error message if you specify a file that does not exist.

You can pass data between chained programs using common memory or data files. Common memory (also called "core common") is a data area in your job's memory area. It can hold a string up to 127 bytes long. You use system function calls to put a string into this area and get it out (see the *RSTS/E Programming Manual*). Data files are described in Part IV of this manual.

Before CHAIN loads a new program, it closes all open files and other I/O channels for the current program. Thus, the new program must open all files and I/O channels it needs to use. CHAIN closes I/O channels implicitly, not explicitly, which means that the last buffer of data is not written to the file. It is therefore advisable to close all files explicitly with a CLOSE statement before chaining to another program; the implicit close can cause the contents of partially filled output buffers or modified virtual array elements to be lost.

Keep the following information in mind when you use CHAIN:

1. It is recommended that you include the file type in the CHAIN statement, especially if you have programs with duplicate file names in your directory. RSTS/E supports many different programming languages under its various run-time systems. If you do not specify the .BAC or .BAS file type, it is possible for the CHAIN statement to load and execute a compiled file that is not a BASIC–PLUS program.

2. The file operations that CHAIN performs can take considerable system time. So use CHAIN only when necessary—that is, only when your programs are too large to fit in memory.

3. When you enter a program through the CHAIN statement, the system sets the STATUS variable. See the *RSTS/E Programming Manual* for more information.

4. On RSTS/E systems with several run-time systems installed, CHAIN can work differently for source and translated programs. For example, suppose you switch to DCL, making it your job keyboard monitor. Then you run a translated BASIC–PLUS program using the DCL RUN command. If that translated program chains to a source program, you will be in the BASIC–PLUS keyboard monitor after the source program is finished executing. On the other hand, if that translated program chains to another translated program, the second translated program returns you to the DCL keyboard monitor when it is finished executing.

5. If a CHAIN statement in a nonprivileged program names a privileged program, the CHAIN statement should not include a line number. You must execute the entire chained program or else the system will not retain the chained program's privilege. (There are some exceptions to this rule—see the *RSTS/E Programming Manual.*)

6. If a CTRL/C is typed while a CHAIN statement is executing, execution halts. The halt always occurs, even if one or both programs have CTRL/C trapping enabled.

PART IV
Data Handling

Chapter 14
Overview of Data Handling

Chapters 9 and 10 introduce data handling. They describe how to enter data into a program with the INPUT and INPUT LINE statements, using your terminal as the input device. They also describe the READ and DATA statements, where the data is contained in the program itself. While these techniques are satisfactory when you need to input a small amount of data, they are inefficient when your program has large amounts of data to process.

Chapters 9 and 10 also show you how to print data on your terminal with the PRINT statement. This technique lets you display the results of your program, but it does not provide a permanent record of its output.

Besides these disadvantages, the I/O methods described so far have another shortcoming. Processing data in a computer is often repetitive. Programs process the same data many times, changing the data each time it is processed. READ, DATA, and simple terminal I/O are not designed for this kind of repetitive processing.

Data files can fill all these needs. They:

- Are an efficient way to store and input data for a program to process
- Provide a permanent record of program input and output
- Allow you to repeatedly process data that changes each time it is processed
- Allow different programs to share the same data

This section of the manual describes how to work with data files in BASIC–PLUS. This chapter covers basic concepts, introduces the three types of BASIC–PLUS files, and describes the OPEN, CLOSE, NAME–AS, and KILL statements, which you use for all types of files. Chapters 15, 16, and 17 describe each type of I/O in detail.

14.1 Files and Devices

A BASIC–PLUS *data file* consists of data to be transferred between programs executing in memory and devices outside memory. The most common peripheral devices are:

- Terminals
- Disks
- Magnetic tape
- Line printers
- Flexible diskettes

Disks and magnetic tape are file-structured devices; the rest are non-file-structured. You can store many different files on a file-structured device. Each file has a file specification that uniquely identifies it. In contrast, you specify only a device name to identify a non-file-structured device like a terminal or line printer. The system treats the entire device as a single "file."

Note that while disks and magnetic tape are file-structured devices, you can also use them as non-file-structured devices on RSTS/E. The *RSTS/E Programming Manual* describes special programming techniques for each type of peripheral device.

14.2 Accessing a File or Device from a Program

To access a file or device from a program, you open a file, read data, write data (or both), and then close the file.

14.2.1 Opening the File

The first step in accessing a file or device is to make it available to your program. You do this by "opening" it with the BASIC–PLUS OPEN statement. In the OPEN statement, you specify (at a minimum) the file or device name and a *channel number*, which must be an integer from 1 to 12.

The OPEN statement makes the file or device available to your program by:

- Creating a logical connection between your program and the file or device, called an *I/O channel.*
- Setting aside an area in memory to be an intermediate storage area between your program and the file or device. This area is called the *I/O buffer* or the *channel buffer.*

The I/O channel is identified by the number you specify in the OPEN statement. You use this channel number to refer to the file or device in other program statements. Channels 1 through 12 are available to your program. The system uses channel 0 for your terminal, so you cannot open a file or device on channel 0.

The I/O buffer is located in your program's memory area; thus, opening a file causes your program to expand. The size of the buffer depends on the type of device you open. The system has a default buffer size for each device; the default buffer size for disks is 512 bytes. You can also specify a buffer size as an optional part of the OPEN statement.

The data in the I/O buffer has the same format as it does on the storage device. In formatted ASCII and virtual array I/O, BASIC–PLUS performs all the processing necessary to make the data available to your program as integer, floating-point, and string data. In block I/O, however, your program controls the buffer directly. You access the data in the buffer as string data, and use conversion functions to make it available to your program as integers or floating-point numbers. You then convert numeric data back to string data for output.

Besides creating an I/O channel and establishing an I/O buffer, the OPEN statement can also perform several other functions. For example, OPEN can create a file with specific characteristics or open a file in a special "mode." Section 14.5 describes the OPEN statement in more detail.

14.2.2 Reading and Writing Data

After you open a file or device, it is available to your program. You can read input from it and write output to it. The statements you use to do this and the amount of processing that BASIC–PLUS performs for you depend on the type of file you choose to work with.

Reading and writing data is a two-step process. When you read data, it moves first from the file to the I/O buffer and then from the I/O buffer to your program. When you write data, the path is reversed. In formatted ASCII and virtual array I/O, BASIC–PLUS handles the buffer. From your point of view, it looks like data moves directly between your program and the file. In block I/O, however, you control both steps in the process.

14.2.3 Closing the File

When you are finished using the file or device, you close it with the CLOSE statement, which closes the I/O channel. This action frees the space taken by the I/O buffer for other use and "disconnects" the file or device from your program. Except when you use block I/O or enter a special form of the CLOSE statement, CLOSE writes the data in the I/O buffer out to the file or device before freeing the buffer space and breaking the logical connection.

The next section introduces the three types of BASIC–PLUS files.

14.3 BASIC–PLUS File Organizations

BASIC–PLUS provides three file organizations:

- Formatted ASCII, also called ASCII stream
- Virtual Array
- Block I/O

You use different programming techniques to work with each type of file. Formatted ASCII files are the simplest files to work with; block I/O files are the most complex. BASIC–PLUS does not support RMS–11 files.

14.3.1 Formatted ASCII Files

If you know how to use the PRINT and INPUT statements for terminal I/O, you already have most of the information you need to work with formatted ASCII files. BASIC–PLUS handles a formatted ASCII file the same way it handles a terminal.

Formatted ASCII files store ASCII characters sequentially in variable-length records. Each record is the same as a line on your terminal: it consists of a line of text followed by a line terminator, which serves as the record delimiter. The record delimiter can be a RETURN (carriage return/line feed), a LINE FEED, a FORM FEED, an ESCAPE, or a VT (vertical tab). A CTRL/Z marks the end of the file. Formatted ASCII files do not usually contain any null characters (ASCII code 0) because the system discards them when it processes formatted ASCII files.

To access a formatted ASCII file, you open it on a channel and then use its channel number in INPUT and PRINT statements. BASIC–PLUS handles the exchange of data between the file, the I/O buffer, and your program, and also performs necessary data conversions between ASCII and numeric data types.

Formatted ASCII files are sequential access files; you cannot access data randomly. You can store formatted ASCII files on disk and tape devices. See Chapter 15 for more information on working with formatted ASCII files.

14.3.2 Virtual Array Files

Virtual arrays are arrays stored on disk. Except for a few differences, which are described in Chapter 16, you work with virtual arrays using same the programming techniques you use for memory arrays.

You can use virtual array files to store arrays that are too large to fit in memory. One file can contain several arrays. Each "record" in a virtual array file is one array element, and virtual arrays can contain real, integer, or string data. You access the data in the file using the same methods you use for memory arrays; no special input or output statements are required. BASIC–PLUS performs the necessary read and write operations to the disk and also manages the I/O buffer.

You can access data in virtual arrays randomly or sequentially, just as you access arrays in memory. You can store virtual arrays only on disk devices. See Chapter 16 for more information on virtual arrays.

14.3.3 Block I/O Files

Block I/O is the most flexible way to do I/O in BASIC–PLUS; it is also the most difficult to program. You will need to learn several new programming techniques to use block I/O.

Block I/O files are sequential or random access files that contain a series of numbered records. Each record in the file corresponds to one physical block on the storage device. You can store block I/O files on disk or tape devices. The record size varies depending on the storage medium.

When you use block I/O, BASIC–PLUS does much less processing for you than when you use formatted ASCII files or virtual arrays. Instead, your program is in control. For example, you:

- Control read and write operations between the I/O buffer and the file
- Operate on the data in the buffer directly
- Define the format and location of data in each record in the file

See Chapter 17 for a complete description of the statements and functions you need to use block I/O.

14.4 Choosing an I/O Method

The I/O method to choose depends on:

- The nature of your application
- The amount of time you want to spend developing the program

Formatted ASCII and virtual arrays provide enough flexibility for many kinds of applications. They are also relatively easy to program because BASIC–PLUS does most of the special processing for you.

Formatted ASCII I/O lets you access all devices the same way—you do not have to know the characteristics of a device in order to use it. Because formatted ASCII files are text files, you can edit them, display them on the terminal, and print them on the line printer. But converting data between ASCII and numeric formats consumes processor time, and ASCII data takes up a lot of storage space. In addition, formatted ASCII files allow sequential access only.

Virtual arrays provide a random access disk file that is easy to use. They also allow you to work with arrays that are too large to be stored in memory. However, the way your program accesses data in the array is very important. To use virtual arrays efficiently, you need to minimize the number of disk accesses; otherwise, your program runs very slowly. Chapter 16 provides the information you need to make efficient use of virtual arrays.

Block I/O provides the additional control and flexibility you need when your application is complex or when speed of execution is very important. Unlike formatted ASCII I/O and virtual arrays, block I/O lets you tailor your program to the characteristics of each device. You can also process any type of logical data record.

BASIC–PLUS provides a set of tools (statements, functions, and special variables) for using block I/O. However, programs that use block I/O are more difficult to develop and debug.

The rest of this chapter describes the basic statements for working with all types of files: OPEN, CLOSE, NAME–AS, and KILL.

14.5 OPEN Statement

The OPEN statement associates a file or device with an I/O channel number. (The I/O channel is a logical entity, having no specific relationship to hardware.)

A channel can be associated with either a file-structured or a non-file-structured device. Consider this example for a file-structured device, such as a disk:

```
OPEN 'FOO.DAT' AS FILE #1
```

An example for a non-file-structured device, such as a terminal, is:

```
OPEN 'KB:' AS FILE #2
```

BASIC–PLUS permits your program to have up to 12 files open at a given time. Channel numbers 1% to 12% can be used with any file or device. (Channel number 0% specifies the terminal owned by your job.)

The general form of the OPEN statement is:

OPEN <string> [{FOR INPUT / FOR OUTPUT}] AS FILE [#] <expression>

The <string> is either a string constant, a string variable, or a string expression that contains a RSTS/E file specification of the form:

dev:[acct]filename.typ/switch(es)

The file name is required for file-structured devices; the device name is required for non-file-structured devices. All other parts of the file specification are optional. By default, the OPEN statement opens a file in your account on the public structure and assigns a protection code of 60. The OPEN statement has no default file type.

After opening a file or device, you perform input and/or output by referring to the channel number. Specify the channel number in the OPEN statement with the integer expression after the keyword FILE. The expression must be an integer from 1 to 12. You cannot open channel 0 because the system opens your terminal on channel 0 when you log in.

Protection codes are normally specified only in the NAME–AS statement, which changes the name and protection code of an existing file (see Section 14.7). However, protection codes can be specified as an optional part of any file specification.

Use the /PROTECT switch to specify the protection code. For example:

```
300 OPEN 'FILE.TYP/PR:40' FOR OUTPUT AS FILE #1%
```

The file FILE.TYP is created with a protection code of 40 and is opened on channel 1.

As in previous versions of RSTS/E, you can also enclose the protection code in angle brackets. However, it is recommended that you use the /PROTECT switch. See the *RSTS/E System User's Guide* for more information.

You can append any or all of the following options to the end of the OPEN statement:

[,RECORDSIZE <expression>] [,CLUSTERSIZE <expression>]
[,FILESIZE <expression>] [,MODE <expression>]

Options must be in the order RECORDSIZE, CLUSTERSIZE, FILESIZE, MODE. If options are out of order, you get the message:

```
?Syntax error
```

Note that except for RECORDSIZE, these options can also be specified as RSTS/E file specification switches. If an OPEN statement contains a switch and an option with the same name, the value specified in the OPEN statement option has precedence over the value specified in the file specification switch.

Omitting an option is the same as specifying that option with a parameter of 0%. In both cases, the option's default value is used. (The MODE option for non-file-structured magnetic tape is an exception; see the *RSTS/E Programming Manual*.)

14.5.1 Forms of the OPEN Statement

The OPEN statement has three distinct forms:

OPEN <string> FOR INPUT AS FILE <expression>
OPEN <string> FOR OUTPUT AS FILE <expression>
OPEN <string> AS FILE <expression>

The form of the OPEN statement determines whether you open an existing file or create a file:

1. An OPEN FOR INPUT statement searches for an existing file (the statement indicates the file is an input file). If the system does not find a file, it returns the error: "?Can't find file or account" (ERR=5).

2. An OPEN FOR OUTPUT statement searches for an existing file and deletes it if found. A new file with the same name is then created. (See the *RSTS/E Programming Manual* for special OPEN modes that can prevent supersession of existing files or that open tentative files.)

3. An OPEN statement without an INPUT or OUTPUT designator attempts to perform an OPEN FOR INPUT operation described in item 1. If this fails, the system creates a new file.

NOTE

The OPEN statement does not restrict a program from performing both input and output operations on a disk file. Nor does it grant read or write access to the file. These privileges are controlled by the file protection code and MODE value. Magnetic tape and DECtape are exceptions; see the *RSTS/E Programming Manual* for more information.

If the program cannot access the file or device, an error is returned. Table 14–1 summarizes common errors that occur on attempted file access.

On DECtape and magnetic tape devices, the FOR INPUT and FOR OUTPUT clauses restrict operations on that file to the operation specified.

When used with disk files, OPEN FOR INPUT and OPEN FOR OUTPUT allow either read or write operations on the opened file. The system allows write access to a file if the protection code permits *and* if no other user has write access to the file.

For example, if user 1 opens a file, he has read and write access. If user 2 opens the same file, he has read access only; a "?Protection violation" error occurs when he attempts to write on that file. When user 1 subsequently closes the file, no user has write access until the next open operation. User 3 can now open the file for both read and write access, because no other user currently has write access to that file.

There are two ways to open a file without gaining write access. You can either specify the /RONLY switch at the end of a file specification or open the file with the Read Only mode. (See the *RSTS/E Programming Manual* for more detail.) Both ways bypass the normal mechanism that grants write access.

You can determine whether the current job has read or write access to a file by testing the STATUS variable immediately after the OPEN statement. The STATUS variable is described in Section 17.1.1.

NOTE

Only one user at a time can have write access to a disk file unless that file is opened in update mode or unguarded mode (MODE value = 1% or 5%).

Table 14–1: OPEN Statement Errors

Value of ERR	Message	Explanation
2	?Illegal file name	The file name specified is not acceptable. It contains unacceptable characters, or else it violates the file specification format.
4	?No room for user on device	The directory space of your device is exceeded, or the device is too full to accept further data.
5	?Can't find file or account	The file or account number is not found on the specified device.
6	?Not a valid device	The device specification is not valid for one of the following reasons: 1. The unit number or its type is not in the system configuration. 2. The logical name has no associated physical device and is thus untranslatable.
8	?Device not available	The specified device exists on the system, but you cannot assign or open it for one of the following reasons: 1. The device is reserved for another job. 2. You lack necessary access privileges for the device. 3. The device is disabled. 4. The device is a keyboard line for pseudo keyboard use only.
10	?Protection violation	You do not have access privileges for the file.
14	?Device hung or write-locked	Check the hardware condition of the device you requested. Possible causes of this error include a line printer out of paper or a high-speed reader off-line.
17	?Too many open files on unit	The system permits only one open output file per DECtape drive and only one open file per magnetic tape drive.
32	?No buffer space available	You accessed a file and the monitor requires one small buffer to complete the request. No small buffer is available.
39	?Magtape select error	When you attempted to access a magnetic tape drive, it was off-line.
46	?Illegal I/O channel	You specified an I/O channel number outside the range of integers 1 through 12.

14.5.2 File-Structured and Non-File-Structured Devices

RSTS/E distinguishes between file-structured devices (disk, DECtape, and magnetic tape) and non-file-structured devices (terminals, for example). You handle them differently when you want to find or create a file.

For a file-structured device, you must include both a device name (or accept the default public disk structure) and a file name in the file specification string of the OPEN statement.

For non-file-structured devices, the device name identifies a file; file name and type, if specified, are ignored. For example:

`KB54:`	Specifies keyboard number 54.
`LP1:`	Specifies line printer unit 1.
`LP1:FILE`	Same as LP1:; the file name is ignored.
`DX1:`	Specifies diskette unit 1.

You need not specify the default device (the public disk storage area).

You can also open a file-structured device in non-file-structured mode. For example:

```
160   OPEN 'DK2:' AS FILE 5%
```

Line 160 opens a disk in non-file-structured mode. The *RSTS/E Programming Manual* describes device-dependent features.

14.5.3 OPEN Statement Options

The next four sections describe the RECORDSIZE, CLUSTERSIZE, FILESIZE and MODE options of the OPEN statement. As these are sophisticated file-handling tools, new users may wish to skip these sections for now and continue with Section 14.6.

14.5.3.1 RECORDSIZE Option

When you open a file, BASIC–PLUS creates an I/O buffer in your memory area to transmit data to and from the file. Normally the device determines the space reserved, because each device has a default buffer size. Table 14–2 lists the default buffer size for each device.

With the RECORDSIZE option, you can allocate more buffer space than is provided by the default case. However, the device may not let your program use additional space.

Table 14–3 shows the buffer sizes you can specify for each device.

The buffer size you specify should be an even number. If you specify an odd number, BASIC–PLUS rounds it down to the next even number.

Table 14–2: Default Device Buffer Size

Device	Default Device Buffer Size (characters or bytes)
Disk	512*
Diskette	512
DECtape	510*
Magnetic tape (DOS or ANSI)	512**
High-speed paper-tape reader	128
High-speed paper-tape punch	128
Line printer	128
Card reader	160
User terminal	128
Null device	2***
DMC11/DMR11	512

*The default buffer size may differ when you use the device as a non-file-structured device.
**For ANSI magnetic tape, the system reads a value from the header label to establish the buffer size.
***See the *RSTS/E Programming Manual* for more information about the null device.

Table 14–3: Use of RECORDSIZE

Device	Possible Buffer Alterations
Disk	A disk may use any buffer size that is an integral multiple of 512 bytes.
DECtape	DECtapes use only the first 510 bytes of the available buffer space (512 bytes for non-file-structured DECtape).
Magnetic tape (DOS or ANSI)	Magnetic tapes use only enough bytes for one physical magnetic tape record. Each physical record must be at least 14 bytes and no larger than the buffer size.
High-speed reader High-speed punch Line printer User Terminal	These non-file-structured devices can use any buffer size greater than the default size. The card reader uses only enough bytes for one card's data.
Diskette	Diskettes use any buffer size that is an integral multiple of 128 bytes.
Null device	The null device can use any even buffer size.
DMC11/DMR11	The DMC11/DMR11 can use any even buffer size. However, you usually make the I/O buffer the same size as the device's receive buffer, which cannot exceed 632 bytes. You can specify the receive buffer size with the FILESIZE option. See the *RSTS/E Programming Manual* for more information.

When you specify a value less than the default, BASIC–PLUS uses the device's default buffer size. To get a buffer size less than the default, specify the desired buffer size, plus 32767%, plus 1%. Smaller buffer sizes are often useful when performing alternate buffer I/O (see Section 17.3.7).

For example, to open a paper-tape reader with a buffer size of two bytes for use with alternate buffer I/O, type:

```
10 OPEN 'PR:' FOR INPUT AS FILE #1%, &
        RECORDSIZE 32767% + 1% + 2%
```

The RECORDSIZE option has significant advantages when you use it with magnetic tape and disk files. On a disk file, you can improve total throughput by using a larger buffer size. This permits a single disk transfer to read a large quantity of data. For example:

```
100 OPEN "MASTER.DAT" FOR INPUT AS FILE 1%, RECORDSIZE 2048%
```

If the file MASTER.DAT occupies a contiguous area on a disk, one 2048–byte transfer takes place. Otherwise, the RSTS/E monitor breaks this quantity into as many as four 512–byte transfers, which takes longer. Either way, the system overhead to perform the transfer is less.

To ensure that a file occupies a contiguous disk area, use the MODE option (see the *RSTS/E Programming Manual*) or the CLUSTERSIZE option described in Section 14.5.3.2.

14.5.3.2 CLUSTERSIZE Option

The CLUSTERSIZE option applies to disk and ANSI magnetic tape files that you create with an OPEN or OPEN FOR OUTPUT statement. The following description applies to disk files. (Refer to the *RSTS/E System User's Guide* for device-specific information and the *RSTS/E Programming Manual* for more details on cluster sizes.)

The RSTS/E system divides each disk into a number of 256–word blocks. Each block is assigned a unique logical block number starting at 1. Block 0 of each disk is reserved for a bootstrap record and is not used by any file. Logical block numbers are assigned such that block n is contiguous with blocks n+1 and n–1.

Contiguous blocks taken together as a unit are called a *cluster*. RSTS/E permits clusters to have 1, 2, 4, 8, 16, 32, 64, 128 or 256 blocks. When the disk is initialized (cleared for use on RSTS/E), a minimum cluster size is established. This minimum cluster size, called the *pack cluster size*, can be 1, 2, 4, 8, or 16 blocks.

To compute the optimal CLUSTERSIZE for a file, divide the file size by 7 and round the result to the next power of 2 (maximum). If you get a result larger than 256, then either use a cluster size of 256 or make the entire file contiguous. The CLUSTERSIZE cannot be less than the minimum, or *pack*, cluster size.

For each file on the system, an entry is made in the User File Directory, or UFD. This entry contains the retrieval information for the file: file name, cluster size, and a sequential list of clusters belonging to that file.

The maximum size of a UFD is seven times its cluster size, which is established when the account is created. The UFD cluster size can be 1, 2, 4, 8, or 16 blocks. A UFD on a disk cannot exceed 112 (decimal) blocks (28,672 words). If files are a minimum size (7 or fewer clusters), a UFD clustered as 16 can hold a maximum of 1157 files. To keep the list of file blocks short, the UFD contains a one-word entry for the first block of each cluster. The first block number of the cluster and the cluster size is sufficient to determine the blocks in the cluster.

Because of the size limit on the UFD, it is a good idea to specify large cluster sizes for large files. In an extreme example, the UFD would be filled by a single file of 24,283 blocks where the file cluster size is one block. However, with a cluster size of 256 blocks, only 128 words of the UFD are required to describe this file.

Omitting the CLUSTERSIZE option is equivalent to specifying CLUSTERSIZE 0%. This assigns a cluster size equal to the pack cluster size for the disk where the file resides. If you attempt to specify a cluster size less than the pack cluster size or if it is not a power of 2, an "?Illegal cluster size" error message (ERR=23) results.

When you specify a negative CLUSTERSIZE for a file, the system uses either the absolute value of the argument specified or the pack cluster size, whichever is greater. A negative cluster size is useful in a program that you plan to run on both large and small systems or on systems that have mixtures of large and small disks. For example, suppose you want to create files with a cluster size of 2. If you specify –2, the program can also create files on disks where the pack cluster size (the minimum cluster size permitted) is greater than 2. For these disks, the system will use the pack cluster size instead of the cluster size you specify.

A sample use of the CLUSTERSIZE option is:

```
100   OPEN "MAT.DAT" FOR OUTPUT AS FILE #1%, CLUSTERSIZE 128%
```

In this example, the file MAT.DAT is created with a cluster size of 128 blocks. MAT.DAT is initially 0 blocks long and is extended as needed in 128–block increments.

The system extends files a cluster at a time. Since clusters are contiguous blocks, you may not find enough contiguous blocks to extend the file (even if enough free, but not contiguous, blocks are available on the disk). If not, the system prints the "?No room for user on device" error message (ERR=4). You should be aware of this when creating a file with a cluster size larger than the *pack* cluster size (the minimum cluster size for that disk).

As another example:

```
100   OPEN "DATA" FOR OUTPUT AS FILE #1%, &
          RECORDSIZE 2048%, &
          CLUSTERSIZE 4%
```

When the system can read or write multiple blocks in a single transfer, the RECORDSIZE option improves disk throughput (see Section 14.5.3.1). By creating the file with a cluster size of 4 (2048 bytes per cluster), you guarantee that virtual blocks 1–4, 5–8, and so forth are contiguous on the disk. These file clusters could be read or written in a single operation.

14.5.3.3 FILESIZE Option

You can preextend a disk file (and only a disk file) by using the FILESIZE option in an OPEN statement. The format for the FILESIZE option is:

OPEN <string> [FOR OUTPUT] AS FILE <expr> ,FILESIZE <expr>

For example:

```
100 OPEN 'VALUES' FOR OUTPUT AS FILE #3%, FILESIZE 50%
```

The data file VALUES is opened and automatically preextended to 50 512–byte blocks.

The argument used with the FILESIZE option must be an integer. To specify a file size of N, where N is between 32768 and 65535 inclusive, specify the FILESIZE argument as 32767% + 1% + (N–32768). This expression converts a signed integer to its unsigned value; that is, its value where bit 15 is 32768 instead of –32768. (See Sections 11.1 and 11.2 for more information on the internal format of integer data.)

You can use the FILESIZE option to preextend files to 65535 blocks only. If you want to preextend a file to more than 65535 blocks, use the /FILESIZE switch in the file specification:

```
100 OPEN 'VALUES/FILESIZE:70000' FOR OUTPUT AS FILE #3%
```

See the *RSTS/E System User's Guide* for more information about the /FILESIZE switch.

You can also use the FILESIZE option on ANSI magnetic tape files, but for a different purpose than described here. See the *RSTS/E Programming Manual*.

14.5.3.4 MODE Option

The OPEN statement allows another option: the MODE field. The format of the OPEN statement, including the MODE field, is:

OPEN <string> [{FOR INPUT / FOR OUTPUT}] AS FILE <expr>, MODE<expr>

The MODE option establishes device-dependent properties of the file. See the *RSTS/E Programming Manual* for device-dependent features.

14.6 CLOSE Statement

The CLOSE statement terminates I/O between a BASIC–PLUS program and a peripheral device. After execution of a CLOSE, BASIC–PLUS reclaims the buffer space assigned to the file and closes the I/O channel between your program and the file. It is good programming practice to close all I/O channels before program execution ends.

The CLOSE statement has the form:

CLOSE [#] <expression> [,[#] <expression> ,...]

The <expression> is the channel number of the file to close. You can close any number of files with a CLOSE statement. To close more than one file, separate the expressions by commas. For example:

```
240 CLOSE #10%
250 CLOSE #2%, #4%
260 CLOSE I% FOR I% = 1% TO 12%
```

Line 240 closes the file open on I/O channel 10. Line 250 closes the files that are open on I/O channels 2 and 4. Line 260 closes all I/O channels available to your program.

You can specify two kinds of CLOSE operations in BASIC–PLUS, a normal CLOSE and a CLOSE with a negative channel number.

14.6.1 Normal CLOSE

To specify a normal CLOSE, simply list the channel numbers of the files to close, as shown in the preceding examples.

The I/O method you use determines the action of a normal CLOSE. In a normal CLOSE of a formatted ASCII or virtual array file, BASIC–PLUS writes the contents of the I/O buffer out to the file before closing it. In block I/O, however, your program controls the I/O buffer; thus, a CLOSE statement does not write the buffer out to the file before closing it.

14.6.2 CLOSE with a Negative Channel Number

When you use a negative channel number in the CLOSE statement, BASIC–PLUS does not write the contents of the I/O buffer out to a formatted ASCII or virtual array file before reclaiming the buffer space assigned to the file. (This is the type of CLOSE that a CHAIN statement performs.)

This type of CLOSE is useful if the last operation you perform on the file (either a read or a write) could place erroneous information in the file or if the last block of the file might not fit on the output device.

A sample program segment that uses a CLOSE with a negative channel number is:

```
10      EXTEND
          .
          .
          .
9000    ON ERROR GOTO 19000
9100    CLOSE CHAN%
9200    GOTO 32767
19000   IF ERL=9100% AND &
           ERR=4% &
           THEN CLOSE -CHAN%
19010   PRINT "END OF FILE HAS BEEN LOST"
19020   RESUME 32767
32767   END
```

This example shows part of a program that writes data to a disk. The disk may be so full that there is no room for the last block of data from the file.

When the CLOSE statement is executed, it is possible that the final buffer of data to be written to the output file might not fit. In this case, program execution goes to line 19000, and the program checks to see if that was the problem. If so, the program performs the CLOSE with a negative channel number to suppress the output of the remaining data. The program then prints a message to notify you that the output file does not contain all the expected data.

You can also use a negative channel number to close a channel where a tentative file has been opened. This type of CLOSE (which is especially useful for temporary work files) deletes the tentative file. See the *RSTS/E Programming Manual* for more information.

14.7 NAME–AS Statement (File Protection and Renaming)

The NAME–AS statement renames or assigns protection codes to a disk file. This statement can be used only by someone who has write access to the file. The format of the statement is:

NAME <string> AS <string>

The file specified in the first string is renamed to the file specified in the second string. When the file is on a device other than the system disk, you must specify the device in the first string and, optionally, in the second string. NAME–AS has no default file type. You must specify the file type in both strings if there is a file type in the old file name or if you want one in the new file name. For example:

```
110   NAME "DM0:OLD.BAS" AS "NEW.BAS"
```

This statement is equivalent to:

```
110   NAME "DM0:OLD.BAS" AS "DM0:NEW.BAS"
```

However, the next statement is not advised because FILE2 has no file type for the system to recognize:

```
190   NAME "FILE1.BAS" AS "FILE2"
```

Use:

```
190   NAME "FILE1.BAS" AS "FILE2.BAS"
```

You can specify a file protection code as part of the second string. Enclose the protection code in angle brackets (<>) or use the /PROTECT switch. (The /PROTECT switch is the recommended method.) If you specify a new file protection code, the new code is assigned to the renamed file. If you do not specify a new protection code, the old protection code is retained. See the *RSTS/E System User's Guide* for a complete description of protection codes.

The following statements both change the protection code of the file FILE.TYP on the system disk to 40. The first statement uses the /PROTECT switch; the second uses angle brackets.

```
200 NAME "FILE.TYP" AS "FILE.TYP/PR:40"
200 NAME "FILE.TYP" AS "FILE.TYP<40>"
```

The next statement changes the name of the file ABC.BAS on DM0:

```
200 NAME "DM0:ABC.BAS" AS "XYZ.BAS"
```

NOTE

You cannot use NAME–AS to change the device or account in which the file resides.

Because you cannot transfer a file from one device to another with the NAME–AS statement, you need not specify the device (DM0: in the previous example) twice. The system generates an error if you specify a device other than the old one.

See Section 5.2.2 for more information about the NAME–AS statement.

14.8 KILL Statement

The KILL statement has the form:

KILL <string>

KILL deletes the file that is named in <string> from your account. You can no longer open this file. If already open, the file remains available until

it is closed. For example, when you complete work with the file XYZ.DAT on the system disk, you can remove the file from storage by executing the statement:

```
460 KILL "XYZ.DAT"
```

You cannot KILL a file that is write-protected against you.

See Section 5.4.2 for more information about the KILL statement.

Chapter 15
Formatted ASCII Input and Output

This chapter describes formatted ASCII I/O, the simplest way to do I/O in BASIC–PLUS. If you know how to use PRINT and INPUT for terminal I/O, you already have most of the information you need to do formatted ASCII I/O to other devices. By combining the PRINT and INPUT statements with the OPEN statement, you can:

- Store formatted ASCII data in disk and magnetic tape files
- Read data from these files
- Do formatted ASCII I/O to line printers or other devices

Be sure to close the file or device with the CLOSE statement when you finish using it.

Formatted ASCII files are sequential access files; they do not allow you to access data randomly or to update data without copying from one file to another.

This chapter reviews the basic forms of the PRINT and INPUT statements, describes their more advanced forms, and shows you how to use them with the OPEN statement.

15.1 PRINT Statement

Earlier chapters show you how to use the PRINT statement in its basic form to print numeric or string data at your terminal. This section reviews this information. It also describes:

- How to use PRINT to write formatted ASCII output to a disk file or other device
- How to tailor the format of output with the PRINT–USING statement, an optional BASIC–PLUS feature

The basic form of the PRINT statement is:

PRINT <list>

This form of the PRINT statement prints a list of values at your terminal. The list can contain any combination of numeric and string values, and each item in the list can be any legal expression. When an item is not a simple variable or constant, BASIC–PLUS evaluates the expression before it prints a value. When you omit the print list, BASIC–PLUS prints a blank line on your terminal.

When printing numbers, BASIC–PLUS:

- Does not print leading zeros or trailing zeros to the right of a decimal point. When a whole number has a decimal point, BASIC–PLUS does not print the decimal point.
- Prints numbers with up to six digits in decimal format and numbers with more than six digits in exponential format.
- Prints all values with six significant digits. To print more than six significant digits, use the PRINT–USING statement.

BASIC–PLUS prints character strings without leading or trailing blank spaces.

Use commas or semicolons between list elements. These characters determine how output is spaced. For example:

```
100    A% = 1% &
       \ B% = 2% &
       \ C% = 3%
110    PRINT A%; A% + B% + C%, C% - A%, "END"
32767 END
RUNNH
 1   6           2             END
```

The comma prints items in print zones. BASIC–PLUS divides a terminal line into print zones of 14 spaces each. (The actual number of print zones is INT (n/14), where n is the size of the print line.) The comma moves the print head to the next available print zone. If the last print zone on a line is filled, the print head moves to the first print zone on the next line. Extra commas make BASIC–PLUS skip print zones.

The semicolon prints items in a packed format. When you use the semicolon between list elements, BASIC–PLUS prints a positive number with a leading and a trailing blank space, a negative number with a leading minus sign and a trailing blank space, and a string exactly as it appears inside quotation marks. No leading or trailing spaces are added. You can omit the semicolon between a string constant and another value. However, its use is recommended for compatibility with BASIC–PLUS–2.

BASIC–PLUS automatically prints a carriage return/line feed at the end of a PRINT statement. To suppress the automatic carriage return/line feed, place a comma or a semicolon at the end of the PRINT list. A comma causes

BASIC–PLUS to start printing in the next print zone; a semicolon causes BASIC–PLUS to start printing in the next blank space.

See Section 9.2.2 for examples of the basic PRINT statement.

15.1.1 Printing Data to a File or Device (Formatted ASCII Output)

To direct output to a device other than your terminal, open the file or device with the OPEN statement and then use the following PRINT statement:

PRINT #<expression>,<list>

The <expression>, which must have the same value as the expression in the OPEN statement, is the channel number of the output file. Its value must be a number from 1 to 12. The <list> can contain one or more variable names, expressions, or constants separated by commas.

For example:

```
100 OPEN 'DATA1.DAT' FOR OUTPUT AS FILE #7%
110 PRINT #7%, 'START OF DATA FILE'
```

These lines open a file called DATA1.DAT on the disk. DATA1.DAT uses channel number 7. The first line in the file reads:

START OF DATA FILE

When you use the PRINT statement to send output to a file, BASIC–PLUS outputs data to the file the same way it prints data on the terminal. The resulting file is called a formatted ASCII or stream ASCII file. Formatted ASCII files are text files; you can display them on your terminal or print them on a line printer.

You can use commas in the PRINT statement to format the data in the file in columns, just as you do when printing data on the terminal. When you use the comma for formatting and print to a device other than a terminal, each line in the file is 72 spaces wide and contains 5 print zones of 14 spaces each.

To write nonprinting ASCII values into the file, use the CHR$ or the STRING$ functions. For example, a CHR$(12%), which is a form feed character, causes line printers and certain types of terminals to skip to the top of the next page. You can force a logical end-of-file by printing a CHR$(26%), which is a CTRL/Z. When you use nonprinting characters, keep in mind the effect they will have when read by a program or sent to an output device.

If you plan to read data from the file with an INPUT statement:

1. Be sure to write commas between data items in the file with string constants. For example:

```
PRINT #1%, NAME$;",";ACCT%;",";AMOUNT
```

2. Be sure that each output line contains a terminator. The INPUT statement expects a terminator (usually a carriage return/line feed) at the end of each line. Remember that ending a PRINT statement with a comma or semicolon produces an output line with no terminator.

3. Do not print records that are more than 132 characters in length, including nulls, line feeds, and carriage returns. The PRINT statement lets you create records with more than 132 characters, but the INPUT statement may not read them correctly.

You can also print data to a file or device with the the PRINT–USING and the MAT PRINT statements, which are described in the next two sections.

15.1.2 PRINT–USING Statement

On systems with the optional PRINT USING feature, you can tailor the format of output with the statement:

```
PRINT [#<expression>,] USING <string>, <list>
```

The <expression>, which is optional, indicates the channel number of the output file. The <string> is either a string constant, string variable, or string expression that is an image of the line to be printed. This string is called the *format field.*

The <list> shows items to be printed in the format specified by the format field. BASIC–PLUS prints characters in the string as they appear, except for the special formatting characters and character combinations described on the following pages. The string (or portions of it) is repeated until the list is exhausted. The rest of this section explains how to construct the format field.

While the examples in this section show terminal output, you can also print to a file or device by including a channel number in the PRINT–USING statement.

15.1.2.1 Exclamation Point

An exclamation point in the format field identifies a one-character string field. This variable string is specified in the PRINT statement list. For example:

```
LISTNH
100   PRINT USING '!!!', 'AB', 'CD', 'EF'
32767 END

Ready

RUNNH
ACE

Ready
```

BASIC–PLUS prints the first character from each of the three string constants or variables. Any characters beyond the first are ignored.

15.1.2.2 String Field

You can specify a string field of two or more characters in the format field with spaces inside backslashes. On some keyboards you can produce the backslash character (\) by typing SHIFT/L. No enclosed spaces indicates a field two columns wide; one enclosed space indicates a field three columns wide, and so on. For example:

```
100    PRINT USING '\\ \ \', 'ABCD', 'EFGHI'
```

This statement prints:

```
AB EFGH
```

The first two backslashes have no enclosed spaces, hence they permit the printing of two characters (AB). The second two backslashes enclose two spaces and permit the printing of four characters (EFGH).

15.1.2.3 Numeric Field

Use the number character (#) in the format field to indicate numeric fields. You can specify any decimal point arrangement this way. Rounding (not truncation) is performed as necessary. For example, this statement prints 12.35 at your terminal:

```
100    PRINT USING '###.##',12.345
```

Also consider the following:

```
100    PRINT USING '####', 12.345
110    PRINT USING '####.', 12.345
120    PRINT USING '##', 100
RUNNH
 12
 12.
% 100

Ready
```

Numeric fields are right-justified. If a number does not fill the allotted space, then spaces precede the number. When the field you specify is too small to print a constant or variable, BASIC–PLUS prints the percent character (%) to indicate the error and then prints the number without reference to the format field.

NOTE

If the numeric field is more than 20 character spaces larger than required to print a constant or variable, a "?Print-using buffer overflow" non-recoverable error may occur.

If the format field specifies a digit preceding the decimal point, at least one digit is always output before the decimal point. If necessary, that digit is zero.

15.1.2.4 Asterisks

If a numeric field designation in the format field begins with two asterisks (**), BASIC–PLUS fills unused spaces in the format field with asterisks. For example:

```
100     A=27.95 &
        \ B=107.50 &
        \ C=1007.50 &
        \ PRINT USING '****.##', A,B,C
RUNNH
**27.95
*107.50
1007.50
```

The asterisks (**) act as additional number characters (#), as well as filling unused spaces.

You cannot use exponential format in a field with leading asterisks. Negative numbers cannot be output using asterisk fill, unless a minus sign follows the number. The use of leading dollar signs ($$) and leading asterisks in the same format field is not compatible with BASIC–PLUS–2.

15.1.2.5 Exponential Format

To print a number in exponential format, use a string that matches the exponential format described in Chapter 8. Use number characters (#) for the number and four circumflex characters (^) for "E±nn", where n is the power expressed in two digits. For example:

```
LISTNH
100     F$='##^^^^'
110     A=10000.
120     PRINT USING F$,A

Ready

RUNNH
10E 03
```

All format positions output a number with an exponent. BASIC–PLUS left-justifies significant digits and adjusts the exponent.

15.1.2.6 Trailing Minus Sign

If you end a numeric format field with a minus sign, BASIC–PLUS prints the sign of the output number after it. A blank space indicates a positive number. For example:

```
LISTNH
100     A=-10.5
110     PRINT USING '##.##- ####.##',A,A

Ready

RUNNH
10.50- -10.50
```

If you do not use the trailing minus sign, you must reserve space in the numeric format field for the sign to precede the number. (An explicit leading minus sign is treated as a literal and therefore is always included.)

15.1.2.7 Dollar Signs

Begin a numeric format field with two dollar signs ($$) to print a dollar sign immediately before the first digit of the number. For example:

```
100     A=77.44 &
        \ B=304.55 &
        \ C=2211.40
110     PRINT USING '$$###.##',A,B,C

Ready

RUNNH
 $77.44
$304.55
% 2211.4
```

The two dollar signs ($$) provide for the printing of one additional digit in the number, preceded by one dollar sign ($).

You cannot use exponential format in a field with leading dollar signs. Furthermore, the floating dollar character cannot output negative numbers unless the minus sign follows the number. The use of leading asterisks (**) and leading dollar signs in the same format field is not compatible with BASIC–PLUS–2.

15.1.2.8 Commas

To print commas in large numbers, insert commas in the numeric format field every three digits to the left of the decimal point. BASIC–PLUS reads a comma to the right of the decimal point as a printing character. For example:

```
100     PRINT USING '*,###,###.## ####.#,#,', 12345.5, 123.456, 1
```

This statement prints:

```
12,345.50 123.5,1
```

15.1.2.9 Insufficient Format

If there are insufficient format characters in a field when a number is output, BASIC–PLUS prints a percent character (%) in the first position of the field, followed by the number in standard format. This usually causes the field to widen to the right. The entire number is printed. For example:

```
100     PRINT USING '##.## ##.##', 12.345, -12.5
```

This statement prints:

```
12.35 %-12.5
```

There is no room in the format to print the minus sign.

When a number has too many decimal places to print using the format field, BASIC–PLUS rounds the number. If the rounded number exceeds the format allowed, BASIC–PLUS prints the percent character (%). For example:

```
100    PRINT USING '.##   .##', .125, .999
```

This statement prints:

```
.13   % .999
```

In this case .999 rounds to 1, which exceeds the format field.

15.1.2.10 Format Too Large

A numeric format field may attempt to output more significant digits than are available. In this instance, zeros substitute for digits that follow the last significant digit. Six significant digits are available with the two-word, single-precision math package. Fifteen digits are available with the four-word, double-precision math package.

When you use the PRINT–USING statement, BASIC–PLUS permits up to 29 formatting characters for single-precision and 19 formatting characters for double-precision. An attempt to print fields larger than 29 or 19, respectively, results in the error message:

```
?Print using buffer overflow
```

In certain cases, BASIC–PLUS truncates a number larger than the format field without an error message. Check for overflow before attempting to output numbers near the maximum size.

15.1.2.11 Formatting and Literal Characters

When you use the PRINT–USING statement, the usual formatting characters (commas and semicolons) have no effect on the output format. The exception is a comma or semicolon at the end of the PRINT list that inhibits termination of the printed line. For example:

```
LISTNH
100    PRINT USING '##   ## ##',1;2,3
200    PRINT USING '#.##', 2.5;
300    PRINT 'X'

Ready

RUNNH
 1    2   3
2.50X

Ready
```

The semicolon at the end of line 200 causes X to print on the same line as 2.50.

Characters that do not have special meanings (letters, for example) are printed as they appear in the PRINT–USING statement. These characters are called *literals*. An example that uses literals in the format string is:

```
100     A=1.32519 &
        \ B=2.45457 &
        \ LET F$ = ' A=##.## B=##.##'
110     OPEN 'LP:' FOR OUTPUT AS FILE #4%
120     PRINT #4%, USING F$, A, B
```

This example prints the following on the line printer:

```
A= 1.33  B= 2.45
```

15.1.3 MAT PRINT Statement

The MAT PRINT statement, one of the optional matrix manipulation statements, allows easy printing of a matrix. The statement has the form:

MAT PRINT [#<expression>,] <matrix name> [, ;]

The optional <expression> is the channel number of the output file.

BASIC–PLUS prints matrices to a file the same way it prints them on a terminal. For example:

```
100     DIM A(16)
120     OPEN 'MAT.DAT' AS FILE #2%
150     MAT PRINT #2%, A(15)
```

This example prints elements 1 through 15 of the matrix A to the file MAT.DAT. One element appears on each line in the file.

If the specified matrix name is unsubscripted, BASIC–PLUS prints the entire matrix (except for the zero elements). Use subscripts to indicate the maximum size of the matrix to be printed.

You can enter a semicolon after the matrix name to indicate that the values are to be printed without additional spaces between them. You can also use a comma to indicate that you want each element printed in its zone. For example:

```
100     DIM A(10,10), B(10,20)
110     MAT PRINT A; &
        ! PRINT MATRIX A IN PACKED FORMAT
120     MAT PRINT B(10,10), &
        ! 10*10 MATRIX PRINTED 5 VALUES PER LINE
```

This example prints a 10–by–10 matrix with no additional spaces between elements and then a widely spaced 10–by–10 matrix.

BASIC–PLUS can also print row and column matrices. For example:

```
10    DIM A(5), B(10)
20    MAT PRINT A; &
      ! PRINT MAT A ON ONE LINE
30    MAT PRINT B &
      ! PRINT IN COLUMN FORMAT

Ready

RUNNH
0   0   0   0   0
0
0
0
0
0
0
0
0
0
0
```

Line 20 prints A as a row matrix, closely packed. Line 30 prints B as a column matrix. Use a comma to print a row matrix with one value per print zone:

```
MAT PRINT A,
```

BASIC–PLUS prints the matrix A as a row matrix (five values per line at your terminal).

15.1.4 PRINT Functions

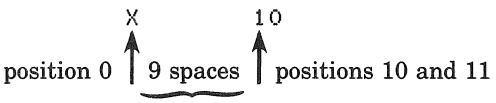
The CCPOS and TAB functions format simple and complex PRINT statements:

Function	Description
CCPOS(X%) or POS(X%)	Returns the current position on the output line, where X% is an I/O channel number in the range 0 to 12. CCPOS(0%) or POS(0%) returns the value for your terminal. Although the two functions are equivalent, CCPOS is preferred for compatibility with BASIC–PLUS–2.
TAB(X%)	Tabs to position X% in the print record. For example, a standard terminal has 72 printable columns numbered 0 through 71. TAB(4%) outputs enough spaces to move the print head to column 4. If the print head is past position 4, no spaces are output.

The following example shows the actions of CCPOS and TAB:

```
100    PRINT 'X'; TAB(10%); CCPOS(0%)
```

This statement prints X at position 0, tabs to position 10 and prints the current position on the output line, which is 10:



The following example shows a use of CCPOS in a program:

```
PRINT IF CCPOS(0%)<>0
```

This statement guarantees that the next PRINT statement will start printing at the left margin. The statement tests the current position of the print head on the terminal. If the current position is at the left margin (that is, CCPOS(0%) = 0), nothing happens. If the current position is anywhere else on the line, the program performs a carriage return/line feed to return the print head to the left margin. This technique is useful for printing prompts or messages at the left margin.

NOTE

CCPOS counts characters. It does not keep track of escape sequences for different types of terminals. Thus, CCPOS may return incorrect results if you are using escape sequences for cursor control.

15.2 INPUT Statement

The INPUT statement enters data from an external device (such as your keyboard or a disk) to a running program. The statement's full form is:

INPUT [#<expression>,] <variable list>

The optional <expression> is the channel number of the input file. See Section 15.2.1 for more information.

When you omit the channel number, your terminal (channel 0) is the input device:

INPUT <variable list>

This form prints a question mark (?) at your terminal. The system then waits for you to respond with the values of string or numeric variables. You can enter one value at a time or you can enter a list of values separated by commas. If you do not type enough values, the system prints another question mark (?). If you type too many values, excess values are ignored.

You can insert printed messages between the variables you wish to input. For example:

```
100   INPUT 'YOUR NAME IS' ; N$, 'ACCOUNT NUMBER';A,'THANK YOU'
```

When this statement is executed, it causes the following interaction at the terminal:

```
RUNNH
YOUR NAME IS? JOE
ACCOUNT NUMBER? 347654
THANK YOU

Ready
```

NOTE

Using more than one prompting message in an INPUT statement is not compatible with other versions of BASIC.

ON ERROR GOTO statements can be used to trap recoverable errors that occur when an INPUT statement is executed. The following errors occur most frequently.

Error	Description	Examples

%Data format error (ERR = 50)	Data input in an illegal form	3.4.5 or $2 or #16 or 2;3 or LORA input for a numeric variable; "HELLO" "THERE" input for a string variable:
		`10  INPUT "TYPE A STRING"; A$`
		`RUNNH`
		`TYPE A STRING? "HELLO" "THERE"`
		`%Data format error at line 10`
		`TYPE A STRING? CAT`
		
		`Ready`

?Illegal number (ERR = 52)	Overflow or underflow	3E+66 or –23
?End of file on device (ERR = 11)	Input CTRL/Z	^Z

BASIC–PLUS assigns values to variables as you input them. You can assign multiple variables by separating them with commas in the INPUT variable list. Similarly, use commas or the RETURN key to separate values as you input them from the keyboard. For example:

```
100   INPUT X,Y,Z
110   PRINT X,Y,Z
RUNNH
? 3.14
? 14,92
 3.14    14    92
```

Do not use commas within a single number; BASIC–PLUS ignores characters input beyond a comma unless another variable is assigned. For example:

Right

```
LISTNH
100    INPUT R &
   \ \ PRINT R

Ready

RUNNH
? 25902
 25902

Ready
```

Wrong

```
LISTNH
100    INPUT R &
     \ PRINT R

Ready

RUNNH
? 25,902
 25

Ready
```

Use quotation marks (") with string variables when you want to preserve embedded commas. For example:

Right

```
LISTNH
100    INPUT M$ &
     \ PRINT M$

Ready

RUNNH
? 'MOUSE, MICKEY'
MOUSE, MICKEY

Ready
```

Wrong

```
LISTNH
100    INPUT M$ &
     \ PRINT M$

Ready

RUNNH
? MOUSE, MICKEY
MOUSE

Ready
```

15.2.1 Reading Data from a File or Device

You can open a file or device for input with the OPEN statement and then use the INPUT statement to make BASIC–PLUS read input from that file or device.

Use the following form of the INPUT statement after an OPEN statement:

INPUT #<expression>,<variable list>

The <expression> must have the same value as the expression used in the OPEN statement. As in the OPEN statement, this expression is the channel number of the input file. Its value must be a number from 1 to 12. The <list> contains one or more variable names separated by commas.

The following example shows the use of OPEN and INPUT. This simple program opens a file, reads three values from it, prints the values on the terminal, and closes the file. The file must be a formatted ASCII file. When BASIC–PLUS reads the file, it interprets the data in the file exactly as if it were typed at the terminal.

```
100 OPEN "DATA.DAT" FOR INPUT AS FILE #1%
120 INPUT #1%,A,B,C
125 PRINT A,B,C
130 CLOSE #1%
140 END
```

To read data from a formatted ASCII file, the INPUT statement that reads the data must match the format of the PRINT statement that wrote the data. Both statements must contain the same number of data items. (Data items in PRINT statements can be constants, variables, or expressions; data items in INPUT statements can only be variables.) The two lists of data items must contain the same data types, in the same order. The following two statements illustrate these rules:

```
100 PRINT #1%, NAME$;",";ACCT%;",";AMOUNT

300 INPUT #2%, CUSTOMER$,ACCTNO%,AMT
```

The PRINT statement writes data to the file. Be sure to write commas between data items in the file if you plan to read the data with INPUT. Remember that INPUT treats the data exactly like terminal input.

The INPUT statement (which may be in a different program) reads data from the file. The variable list matches the PRINT statement's variable list in both type and order. Note that if you print data to a file and want to read it later in the program, you must close the file and reopen it before you can read data from it.

Each INPUT statement reads data from one record in the file. A record is a series of ASCII characters up to a line delimiter (RETURN, FORM FEED, LINE FEED, or ESC). It is recommended that you limit the length of each record to 132 characters, including line delimiter characters. The INPUT statement may not always read longer records correctly.

Commas separate individual values in a record. If there are more values in the record than there are variables in the INPUT statement, BASIC–PLUS ignores any extra values (as it does with terminal input). However, if the record contains too few values, BASIC–PLUS returns the error "?Not enough data in record" (ERR = 59). (This action differs from terminal input, where the INPUT statement prompts for additional values.)

When an INPUT statement encounters a CTRL/Z, BASIC–PLUS returns the error "?End of file on device" (ERR = 11). Include an error handling routine in your program to process this error.

You can also read data from a formatted ASCII file with the INPUT LINE and MAT INPUT statements. See Sections 15.2.3 and 15.2.4.

15.2.2 Opening Your Terminal as an I/O Channel

There are two ways to open your terminal as an I/O channel in BASIC–PLUS:

1. You can use OPEN and INPUT to open your terminal on a nonzero channel. For example:

```
100 OPEN 'KB:' FOR INPUT AS FILE #2%
110 INPUT #2%, A
```

   BASIC–PLUS does not print a prompting question mark at the terminal. For example:

```
100 OPEN 'KB:' FOR INPUT AS FILE #2%
110 INPUT #2%, A
120 PRINT A
130 END
```

   When you run this program, it pauses to let you enter a value for A, but does not prompt for the value with a question mark:

```
RUNNH
567.8
 567.8
```

   Use this method to open your terminal in a special mode. Do not include a prompting message in this type of INPUT statement. See the *RSTS/E Programming Manual* for information on special terminal operations.

2. You can use the INPUT statement by itself to access your terminal on channel 0. (You cannot specify a channel number of 0 in an OPEN statement.) For example:

```
10 INPUT #0%, A$
```

   This statement is the same as:

```
10 INPUT A$
```

   When you specify channel 0, BASIC–PLUS prints the prompting question mark. You can include a prompting message in this type of INPUT statement.

15.2.3 INPUT LINE Statement

The INPUT LINE statement lets you enter a line of data as a single character string, regardless of embedded spaces or punctuation. This is different from normal string input, where the comma, apostrophe, and single and double quotation marks have special meanings.

INPUT LINE has the format:

INPUT LINE [#<expression>,] <string variable>

For example:

```
150 INPUT LINE A$
```

When this statement is executed, the program pauses to let you enter a line followed by a RETURN, FORM FEED, LINE FEED, or ESC (see Section 10.3). You can enter a line up to 132 characters long, including line delimiter characters.

BASIC–PLUS reads every character you type into the variable A$, including quotation marks, commas, and the line delimiter (for example, carriage return/line feed). To remove the line delimiter, use the CVT$$ function or the LEFT$ function, which are described in Section 10.5. When you input CTRL/Z, the system returns the error "?End of file on device" (ERR = 11). Include an error handling routine in your program to process this error.

The INPUT LINE statement can read data from a file as well as from a terminal. The following example opens file FZ.DAT on channel 7 and reads a string of characters up to the next terminating character:

```
100 OPEN 'FZ.DAT' FOR INPUT AS FILE #7%
110 INPUT LINE #7%, B$
```

To ensure that INPUT LINE reads the string correctly, it is recommended that you limit its length to 132 characters, including line delimiter characters.

When reading data from a file, the INPUT LINE statement treats a carriage return character (ASCII code 13 decimal) differently depending on what character follows it. When the line feed character (ASCII code 10 decimal) follows the carriage return character, these two characters delimit (end) the line. The INPUT LINE statement returns a string that ends with a carriage return/line feed.

When a null character (ASCII code 0) follows the carriage return, the INPUT LINE statement discards both the carriage return character and the null character and continues processing the string.

When any other ASCII character follows the carriage return character, the INPUT LINE statement treats the carriage return character as a normal data character instead of a line delimiter.

15.2.4 MAT INPUT Statement

The MAT INPUT statement, one of the optional matrix manipulation statements, inputs the values of a matrix from a specified input channel. When you do not specify a channel, your terminal is used. For example:

```
200   MAT INPUT A(20)
```

Line 200 reads 20 floating-point values as elements of the matrix A. The MAT INPUT statement has the form:

MAT INPUT [#<expression>,] <list of matrices>

BASIC–PLUS reads input from a file or device open on the channel indicated by the expression.

The following lines open the file DATA1 on DM1: on channel 1 (of 12 possible channels), and read a matrix of values to fill B(10,25):

```
140    DIM B(10,25)
200    OPEN 'DM1:DATA1' FOR INPUT AS FILE #1%
210    MAT INPUT #1%, B
```

The zero elements are not assigned a value. When the input channel is your terminal (channel 0), BASIC–PLUS prints a question mark (?). However, reference to another channel does not print the prompting character. Depending on the name of the matrix, the MAT INPUT statement allows input of floating-point, integer, or string values.

15.3 Formatted ASCII Examples

The following programs illustrate formatted ASCII input and output. The first program creates a disk file, prompts the user to enter customer names and addresses at the terminal, and writes the names and addresses into the disk file. The second program reads the data from this disk file and prints it on the terminal.

The first program:

- Does not supersede an existing file. The program has an error handling routine that prompts the user for a new file name if the name entered already exists. The program also opens the new file MODE 128%, which tells the system not to supersede an existing file. (See the *RSTS/E Programming Manual* for information about MODE values.)

- Uses INPUT LINE statements to read input from the terminal. Use of INPUT LINE lets the program read values that contain embedded commas. The program then uses the CVT$$ function to strip off the line terminators that the user types.

- Writes the data into the file with PRINT statements. Each record in the file contains either a customer name, a street address, or a city and state.

- Prompts the user after each entry to find out if there is more data to be entered. The user is prompted to answer "YES" or "NO". The program accepts an answer in either uppercase or lowercase. It uses the CVT$$ function to convert whatever the user types into an uppercase value.

```
50  EXTEND
100 PRINT "Enter name of customer file to be created";
110 INPUT LINE CUSNA$    !Get name of file from terminal &
                         !Use INPUT LINE in case the name &
                         !includes a PPN, which has an &
                         !embedded comma
120 CUSNA$ = CVT$$(CUSNA$,4%) &
                         !Strip off line terminators
125 ON ERROR GOTO 19000
130 OPEN CUSNA$ FOR OUTPUT AS FILE #1%, MODE 128% &
                         !Open file only if it doesn't &
                         !already exist (MODE 128%)
200 !Now prompt terminal operator for name and address &
    !information and write it to the name and address file.
210 PRINT "Enter Customer Name"; &
    \ INPUT LINE CUST.NAME$      !Get whole name &
    \ CUST.NAME$ = CVT$$ (CUST.NAME$,4%) &
                                 !Strip off line terminators
220 PRINT "Enter Street Address"; &
    \ INPUT LINE CUST.ADDR1$     !Get first address line &
    \ CUST.ADDR1$ = CVT$$(CUST.ADDR1$,4%) &
                                 !Strip off line terminators
230 PRINT "Enter City and State"; &
    \ INPUT LINE CUST.ADDR2$     !Get second address line &
    \ CUST.ADDR2$ = CVT$$(CUST.ADDR2$,4%) &
                                 !Strip off line terminators
300 !Now write customer name and address information &
    !to the output file.
310 PRINT #1%, CUST.NAME$        !Print the customer's name
320 PRINT #1%, CUST.ADDR1$       !Print first line of address
330 PRINT #1%, CUST.ADDR2$       !Print second line of address
400 ANYMORE$ = ""                !Clear answer string
410 PRINT "Any more names and addresses? (Answer YES or NO)" &
    \INPUT ANYMORE$ &
    !Ask operator if there are any more customers to be &
    !added to the name and address file
420 ANYMORE$ = CVT$$(ANYMORE$,511%) &
    !Convert lowercase answer to uppercase, remove leading &
    !spaces and tabs, etc.
430 GOTO 200 IF ANYMORE$ = "YES" !Loop back if more
440 IF ANYMORE$ <> "NO" THEN GOTO 410 &
    !If user enters incorrect answer, prompt again for yes or no
450 CLOSE #1%                     !Terminate if no more
460 GOTO 32767
19000 !Error handling routine
19010 IF ERR = 16% AND ERL = 130 THEN &
      PRINT "File already exists - enter a new file name" &
      \ RESUME 110 &
      !If file already exists, tell user and retry
19999 RESUME 0  !Let BASIC-PLUS handle all other errors
32767 END
```

The next example reads the names and addresses from the customer file and prints them on the terminal. The program:

- Uses INPUT LINE to read the data so that embedded commas are handled properly.

- Uses PRINT statements to print each record in the file on the terminal.

- Contains an error handling routine to process the errors "?Can't find file or account" (ERR = 5) and "?End of file on device" (ERR = 11). The first error occurs when the user enters a file name that does not exist; the second error occurs when the INPUT LINE statement encounters a CTRL/Z, which marks the end of the file.

```
50 EXTEND
100 PRINT "Enter name of customer file to be printed";
110 INPUT LINE CUSNA$    !Get name of file from terminal &
                         !Use INPUT LINE in case name &
                         !contains embedded comma
120 CUSNA$ = CVT$$(CUSNA$,4%) &
                         !Strip off line terminators
130 ON ERROR GOTO 19000
140 OPEN CUSNA$ FOR INPUT AS FILE #2% &
                         !Open file if it already exists &
                         !Use default RECORDSIZE
150 !Now read name and address of one customer, &
    !which is stored in three records, one for name, &
    !one for street address, and one for city and state. &
    !Use INPUT LINE so embedded commas are handled properly.
160 INPUT LINE #2%, CUST.NAME$ &
    \ CUST.NAME$ = CVT$$(CUST.NAME$,4%) &
    !Read name and strip off line terminators
170 INPUT LINE #2%, CUST.ADDR1$ &
    \ CUST.ADDR1$ = CVT$$(CUST.ADDR1$,4%) &
    !Read street address and strip off line terminators
180 INPUT LINE #2%, CUST.ADDR2$ &
    \ CUST.ADDR2$ = CVT$$(CUST.ADDR2$,4%) &
    !Read city and state and strip off line terminators
190 PRINT CUST.NAME$ &
    \ PRINT CUST.ADDR1$ &
    \ PRINT CUST.ADDR2$ &
    \ PRINT              !Print name, address, and a blank line
200 GOTO 160             !Loop back and read next name and address
19000 IF ERR = 5% AND ERL = 140 THEN &
      PRINT "The file name you entered does not exist." &
      \ RESUME 100 &
      !If file does not exist, tell user and reprompt for file name.
19010 IF ERR = 11% AND ERL = 160 THEN &
      PRINT "Processing is complete" &
      \ CLOSE #2% &
      \ RESUME 32767 &
      !End of file reached on input, print completion message, &
      !close file, and end program.
19999 RESUME 0      !Let BASIC-PLUS handle all other errors
32767 END
```

Chapter 16
Virtual Arrays

Many applications require you to address and update individual records on a disk file in a random manner. Other applications require more room for storing arrays than is economical in main memory. BASIC–PLUS fills both requirements with a random access file system called *virtual arrays*.

A virtual array is a data array that is stored in a disk file. By using virtual arrays, you can:

- Store arrays in a permanent form
- Operate on arrays that are too large to fit in memory

One virtual array file can contain several data arrays.

To use virtual arrays, you combine the programming techniques for memory arrays with the programming techniques for file access. For example, to set up a virtual array, you use a special form of the DIM statement to dimension the array, and you open a file. You can then reference any element of one or more arrays in the file the same way you reference memory arrays. No explicit I/O statements are needed. BASIC–PLUS automatically reads data from the file into the I/O buffer, transfers data between the buffer and your program, and writes data from the buffer to the file. When you finish using a virtual array, you close the file.

While virtual arrays are similar to memory arrays, they are not identical. Unlike memory arrays, for example, virtual array elements are not set to 0 when your program starts executing. Instead, when you open a virtual array file, elements have whatever value happens to be on the disk. You must initialize the array in your program. String handling also differs in memory arrays and virtual arrays.

Because virtual arrays are disk files, the order in which you reference array elements can significantly affect execution time. BASIC–PLUS reads portions of the array from disk into the I/O buffer in response to references in your program. This chapter describes how virtual arrays are stored and explains how to reference array elements with a minimum number of disk accesses.

16.1 Virtual Array DIM Statement

To have a matrix of data in a virtual array, you must declare it in a special form of the DIM statement. This special DIM statement is:

DIM #<integer constant>, <list>

The integer constant is between 1 and 12. This integer constant corresponds to the channel number where the program has opened a disk file. The variable list appears as it would in a DIM statement for an array in main memory. Thus, a 100–by–100 array of floating-point numbers is defined as:

```
100     DIM #12%, A(100,100)
```

You can store floating-point constants, integer constants, and strings in virtual array matrices. You can also specify more than one matrix in a virtual array file. For example:

```
250     DIM #1%, A(1000), B%(2000), C$(2500)
```

Line 250 allocates space for 1001 floating-point numbers, 2001 integer numbers, and 2501 16–character strings (note that space for the 0th element is included for each matrix). However, if you define a virtual array in this fashion, you should dimension the arrays to the same size with future references.

16.2 Virtual Array String Storage

String data is handled differently in virtual arrays than in memory arrays. In memory arrays, you do not specify string length. Instead, it varies dynamically between 0 and 32767 characters. In virtual arrays, however, strings have a fixed maximum length. The maximum length must be one of the following powers of 2:

2, 4, 8, 16, 32, 64, 128, 256, 512

You specify the maximum length for virtual array strings in the DIM statement. Use the form:

DIM #<integer constant>,<stringvar(dimension(s))>[= <constant>]

For example:

```
150 DIM #1%, A$(100) = 32%, B$(100) = 4%, C$(100)
```

This statement dimensions a virtual array file that contains three string arrays:

A$ Consists of 101 strings of 32 characters each, maximum.

B$ Consists of 101 strings of 4 characters each, maximum.

C$ Consists of 101 strings of 16 characters each, maximum.

Each element in a string array has the same maximum length.

Although elements can be shorter than the maximum length, BASIC–PLUS reserves space in the file for each element to be the maximum length. If you do not specify a maximum length, BASIC–PLUS assumes a default length of 16 characters.

If you specify a length that is not a power of 2, BASIC–PLUS uses the next higher size. For example, these two statements have the same result:

```
100 DIM #1%, X$(10) = 65
100 DIM #1%, X$(10) = 128
```

Both statements set the maximum string length of X$ to 128 characters.

In addition to string length, BASIC–PLUS also handles trailing null characters differently in memory arrays and virtual arrays. BASIC–PLUS retains trailing nulls for strings in memory, but drops them when accessing string data in virtual arrays. (See Section 16.4.1 for more information.)

16.3 Opening and Closing a Virtual Array File

To reference your virtual array file, you must first associate a disk file name with a channel number from 1 to 12. (You must also use this same channel number in the virtual DIM declaration.) Do this with an OPEN, OPEN FOR INPUT, or OPEN FOR OUTPUT statement:

OPEN <string> [{ FOR INPUT / FOR OUTPUT }] AS FILE [#] <expression>

The string is the disk file name and the expression specifies the channel number. (This format is described in Section 14.5.) For example:

```
350     OPEN 'ACCT.DAT' AS FILE #1%
```

This statement associates the file named ACCT.DAT with I/O channel 1. If ACCT.DAT exists, the system uses the existing file. If there is no file named ACCT.DAT, the system creates one. If you wish to destroy a file named ACCT.DAT and create a file of the same name, you can use the statement:

```
350     OPEN 'ACCT.DAT' FOR OUTPUT AS FILE #1%
```

This statement deletes the existing file and creates a new one. If you want the system to alert you that the file ACCT.DAT is not present, type:

```
350     OPEN 'ACCT.DAT' FOR INPUT AS FILE #1%
```

The system returns an error message if ACCT.DAT is not found:

```
?Can't find file or account at line 350
```

BASIC–PLUS does not initialize virtual arrays when you open them. Thus, each element contains whatever value happened to be on the disk when the file was created. For this reason, immediately after creating a virtual array, you should:

1. Set all elements of numeric arrays to zero
2. Set all elements of string arrays to the null string ("")

Virtual arrays permit internal buffers larger than 512 characters. Therefore, you can use the RECORDSIZE option when opening a virtual array file. If specified, the RECORDSIZE must be one of the following powers of 2: 512, 1024, 2048, 4096, 8192, or 16384.

16.3.1 Preextending a Virtual Array

The system overhead for extending a file by one data element or by many elements is nearly the same. Thus, it is more efficient to immediately extend a new file to its final length than to extend it many times. Whenever you know the maximum size of a file, you should extend it to its full size. For example:

```
100     DIM #1%, A(10000%)
110     OPEN 'DATA' FOR OUTPUT AS FILE #1%
120     A(10000%)=0
```

This example extends the virtual array A to its final length. However, BASIC–PLUS does not initially zero virtual arrays. In the previous example, A(0) through A(9999) contain indeterminate values. Unless you are careful, these values could cause a program failure. You should first zero the virtual array as follows:

```
300     A(I%)= 0.0 FOR I% = 10000% TO 0% STEP-1%
```

This statement immediately extends the file to its final size and then zeros it sequentially. The following statement performs the same function but is less efficient:

```
300     A(I%)= 0.0 FOR I% = 0% TO 10000%
```

This statement extends the file each time it requires another disk block.

16.3.2 Closing a Virtual Array File

The CLOSE statement terminates I/O between the BASIC–PLUS program and the virtual array. Once you close a virtual array, you can reopen it for reading or writing on any channel.

You must close virtual arrays before the end of program execution. A CLOSE statement with a positive channel number outputs the last buffer of data elements to the virtual array file. A CHAIN statement or a CLOSE with a negative channel number automatically closes open virtual arrays but does not output the contents of the I/O buffer to the array. The format of the CLOSE statement is:

CLOSE [#] <expression> [,[#] <expression> ,...]

The expression has the same value as the expression in the OPEN statement. It indicates the channel number of the array to close. You can close any number of arrays with a single CLOSE statement. To close more than one array, separate the expressions with commas. For example:

```
255       CLOSE #2%,#4%
345       CLOSE #10%
495       CLOSE #-5%
```

Line 255 closes the virtual arrays opened on channels 2 and 4. Line 345 closes the array open on channel 10. Line 495 closes channel 5 without writing the I/O buffer to the disk file.

16.4 Virtual Array Programming Conventions

When you use virtual arrays, you can encounter recoverable errors if your program does any of the following:

- References a virtual array without first opening the file
- References a nondisk file (for example, a magnetic tape file or a line printer) as a virtual array
- Defines an array bigger than the available disk storage on the system

Remember that you must close a virtual array file before stopping the program or chaining to another program.

16.4.1 Virtual Array Storage

Any data element in a virtual array is contained in a single block (512 bytes) of disk storage. This restriction has no effect on integers and floating-point items, where the size of data items is fixed. However, it limits the maximum length of a virtual array string element to 512 characters. The number of data elements stored in each disk block depends on the size of each element.

The number of elements in virtual strings depends on the maximum string length you specify in the DIM# statement. The size of a virtual string defaults to 16 characters; you can specify 2, 4, 8, 16, 32, 64, 128, 256, or 512 characters. Table 16–1 indicates the number of array elements stored in each block of a virtual array file.

Table 16–1: Virtual Array Storage

Data Type	Number of Elements per Block
Integer (%)	256
Two-Word Floating-Point	128
Four-Word Floating-Point	64
String ($) (N equals the maximum length)	512/N

Strings in virtual arrays occupy preallocated space in the virtual file. They differ from strings in memory, where space is allocated dynamically. A disk block containing virtual strings can be considered to be a succession of fields, each having the maximum string length.

When a virtual string is assigned a new value, it is stored left-justified in the appropriate field. If the new string value is shorter than the maximum length, the remainder of the field is filled with trailing null characters. When the string is retrieved, its length is computed as the maximum string length minus the number of trailing null characters.

16.4.2 Translation of Array Subscripts into File Addresses

When you reference a virtual array element, BASIC–PLUS translates it into a file address. To translate an array subscript into a file address, BASIC–PLUS computes the relative distance from the specified item to the first item in the array. Then it adds the relative distance from the first element of the array to the first item in the file. The first quantity is computed from the array subscript and the number of elements per block, as shown in Table 16–1. The second number is a constant for each array in a file and is computed from the parameters specified in the DIM# statement.

Since the DIM# statement contains the only information used to define the structure of a file, you can specify different accessing arrangements for the same file in one or more programs. For example, you can reference the same data as either a series of 32–byte strings (A2$) or 16–byte strings (A1$):

```
10      DIM #1%,A1$(1001) = 16          !16 CHARACTER STRINGS
20      DIM #1%,A2$(500) = 32           !32 CHARACTER STRINGS
30      OPEN 'FIL1.DAT' AS FILE #1%           !VIRTUAL ARRAY FILE
```

Remember, in BASIC–PLUS array subscripts begin with 0 and not 1. An array with dimension n or (n,m) contains n + 1 or [(n + 1)*(m + 1)] elements.

Your program can define two-dimensional virtual arrays as well as one-dimensional virtual arrays. Two-dimensional arrays are stored linearly, row-by-row, on disk and in memory. Thus, in the case of an array X(1,2), the array appears logically as follows:

X(0,0)	X(0,1)	X(0,2)
X(1,0)	X(1,1)	X(1,2)

Physically this array is stored as follows:

X(0,0)	Lowest Address
X(0,1)	
X(0,2)	
X(1,0)	
X(1,1)	
X(1,2)	Highest Address

When you reference a virtual array sequentially, it is more efficient to reference the rows, rather than the columns, in sequence. For example, Program 1 computes the sum of each row and column in a two-dimensional array far more efficiently than Program 2.

Program 1

```
LISTNH
100    REM - PROGRAM 'ONE' TO COMPUTE SUMS EFFICIENTLY &
             'ARRAY' CONTAINS VIRTUAL ARRAY &
             R(I%) IS SUM OF ROW I &
             C(J%) IS SUM OF COLUMN J
110    DIM #1%,A(10%,50%) &
       ! 10 ROWS, 50 COLUMNS
120    DIM R(10%), C(50%) &
130    OPEN 'ARRAY' FOR INPUT AS FILE #1% &
       ! OPEN VIRTUAL FILE AND INITIALIZE SUMS WITH MAT
140    MAT R = ZER &
       \ MAT C = ZER
150    FOR I% = 1% TO 10% &
       ! OPERATE ROW-BY-ROW
160    FOR J% = 1% TO 50% &
       ! DO EACH COLUMN IN ROW
170    R(I%) = R(I%) + A(I%,J%) &
       ! TOTAL ACROSS ROW
180    C(J%) = C(J%) + A(I%,J%) &
       ! TOTAL DOWN COLUMN
190    NEXT J% &
       \ NEXT I% &
       ! COLUMN SUM IS INSIDE LOOP
200    MAT PRINT R; &
       \ MAT PRINT C; &
       ! PRINT ROW TOTALS, THEN COLUMN TOTALS
210    CLOSE #1%
32767  END

Ready
```

Program 2

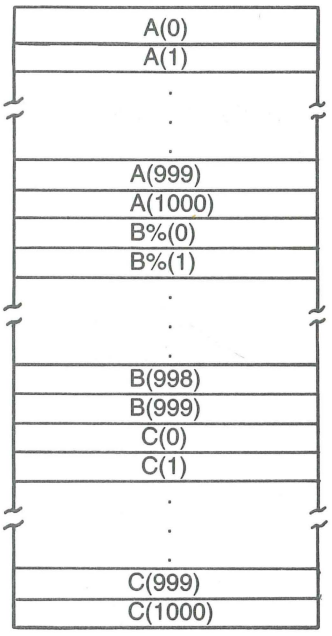
```
LISTNH
100    REM - PROGRAM 'TWO' USES VIRTUAL MEMORY INEFFICIENTLY
110    DIM #1%,A(10%,50%) &
       ! 10 ROWS, 50 COLUMNS
120    DIM R(10%), C(50%) &
130    OPEN 'ARRAY' FOR INPUT AS FILE #1% &
       ! OPEN VIRTUAL FILE AND INITIALIZE SUMS WITH MAT
140    MAT R = ZER &
       \ MAT C = ZER
150    FOR J% = 1% TO 50% &
       ! OPERATE ONE COLUMN AT A TIME
160    FOR I% = 1% TO 10% &
       ! AND ACROSS ROW
170    R(I%) = R(I%) + A(I%,J%) &
       ! TOTAL ACROSS ROW
180    C(J%) = C(J%) + A(I%,J%) &
       ! TOTAL DOWN COLUMN
190    NEXT I% &
       \ NEXT J%
200    MAT PRINT R; &
       \ MAT PRINT C; &
       ! PRINT ROW TOTALS, THEN COLUMN TOTALS
210    CLOSE #1%
32767  END

Ready
```

In virtual arrays, two or more arrays can share the same file. For example, the following DIM# statement is legal:

```
100      DIM #1%,A(1000),B%(999),C(1000)
```

Array B% begins immediately after element 1000 of A and array C begins immediately after B%(999). The disk layout is shown in Figure 16–1.

Figure 16–1: Virtual Array File Layout



Because arrays are stored sequentially on disk, you should not set up logical records that consist of matching elements of different arrays. The following example, which prints a mailing list, shows an inefficient use of multiple arrays in a virtual array file:

```
100 DIM #1%, FIRST.NAME$(100%), LAST.NAME$(100%),&
    ADDRESS$(100%) = 64%
150 OPEN "MAIL.VIR" FOR INPUT AS FILE #1%
200 FOR I% = 1% TO 100%
210 PRINT FIRST.NAME(I%);" ";LAST.NAME$(I%)
220 PRINT ADDRESS$(I%)
230 PRINT !Blank line
240 NEXT I%
250 CLOSE #1%
32767 END
```

In this example, several disk accesses are needed to print each individual's name and address.

The following program uses a single array element as a logical record. This program is more efficient than the first one because only one disk access is needed to print each individual's name and address.

```
100 DIM #1%, CUSTOMER$(100%) = 128%
150 OPEN "MAIL.VIR" FOR INPUT AS FILE #1%
200 FOR I% = 1% TO 100%
210 PRINT LEFT(CUSTOMER$(I%),16%);" "; &
    PRINT MID(CUSTOMER$(I%),16%,16%)
220 PRINT MID(CUSTOMER$(I%),33%,64%)
230 PRINT !Blank line
240 NEXT I%
250 CLOSE #1%
32767 END
```

For string arrays in virtual array files, you should specify the maximum length of the longest element of that array. This is because BASIC–PLUS allocates a fixed number of bytes in the disk file to elements in string arrays (see Section 16.2).

A single string element never crosses a disk block boundary, where each disk block contains 512 bytes or 256 words. For example:

```
100      DIM #1%,A%(2),B$(1000)=4
```

BASIC–PLUS allocates the first three words of the disk block to A%. If array B$ were to begin immediately after A%, one of the elements of B$ would cross a block boundary. Hence, B$ begins at the start of the second block in the file rather than immediately after A%.

When you assign more than one array to a single virtual array file, each array begins immediately after the last element of the preceding array. This occurs unless such an allocation splits an element of the array across two disk blocks. To avoid this split, the array begins at the start of the next block of the file. The remaining words in the first block are not used.

16.4.3 Access to Data in Virtual Arrays

Only a portion of a virtual array is in memory at once. BASIC–PLUS transfers this data directly between the disk and an I/O buffer in your area. BASIC–PLUS creates the I/O buffer when you execute the OPEN statement. This buffer must be a multiple of 512 bytes (one block) long; you may specify it as several blocks with the RECORDSIZE option in the OPEN statement. For each virtual array file, BASIC–PLUS notes:

1. Which block of the file is in the buffer
2. Whether the data in the buffer has been modified since it was read into memory

After BASIC–PLUS translates a virtual array address into a file address, it checks whether the block containing the referenced item is in the buffer. If the necessary block is present, the reference proceeds; if not, BASIC–PLUS reads another portion of the file into the buffer. If you alter the current data in the buffer, BASIC–PLUS rewrites this data on the disk before reading new data into the buffer.

BASIC–PLUS ultimately locates all references to virtual arrays by using file addresses relative to the start of the file. No symbolic information concerning array names, dimensions, or data types is stored in the file. Thus, different programs can use different array names to refer to the data in a single virtual array file. Be cautious in such operations; it is your responsibility to ensure that programs referencing a set of virtual arrays are referencing the same data. For example:

Program 1 contains:

```
100     !PROGRAM ONE
110     DIM #1%, X(10), Y(10)
120     OPEN 'FILE.DAT' AS FILE #1%
```

Program 2 contains:

```
100     !PROGRAM TWO
110     DIM #1%, Z(10), X(10)
120     OPEN 'FILE.DAT' AS FILE #1%
```

Whenever Program 2 references array Z, it uses the data known to Program 1 as array X. Both X and Z are the first arrays in their declarations, both contain floating-point data, and both are 11 elements (X(0),...,X(10)) long. These two arrays, then, correspond in position, type, and dimension.

References to the array X in Program 1 and to the array X in Program 2 do not refer to the same data, even though both use the virtual file FILE.DAT.

Note also that the two statements in Example 1 are not equivalent to the statement in Example 2:

Example 1

```
100     DIM #1%,A(10%)
110     DIM #1%,B(10%)
```

Example 2

```
100     DIM #1%,A(10%), B(10%)
```

In Example 1, the arrays A and B equal each other; they constitute the first array in the file open on channel 1. In Example 2, both arrays A and B exist in separate areas in the file open on channel 1.

Be careful not to open a file under two channel numbers. For example:

```
150     OPEN 'VALUES' AS FILE #1% &
        \OPEN 'VALUES' AS FILE #2%
          .
          .
          .
200     DIM #1%, X$(20%)
210     DIM #2%, Y$(20%)
```

This example creates two buffers for the storage of channel 1 and channel 2 I/O. If you change the same block of the file in both buffers, the system adds the changes from only one of the buffers to the file. The system may overwrite the buffer you wrote first with the other buffer. Consequently, any data change in the first buffer is lost.

If you attempt to write on file #2, you get a "?Protection violation" error, unless the file is open in update mode (see the *RSTS/E Programming Manual*). In either case, the write is unsuccessful.

16.4.4 Allocating Disk Storage to Virtual Array Files

The dimensions in a DIM# statement set maximum values for subscripts. They do not compute the initial size of the virtual array file allocated on disk. Instead, the system creates the file with a length of zero blocks and appends blocks to the file for the highest file address referenced in the array.

You can thus specify array dimensions larger than required when you write a program. Such programs can operate on larger arrays without modification and without tying up disk storage. The only place you find areas of unallocated disk storage is at the end of the file.

As you append blocks to a file, the system does not initialize their contents to zero. The data previously recorded in a block is available to the new owner of the block. If your files contain confidential information, you should overwrite the data before deleting the file.

To override the dynamic virtual array allocation, you can reference the last element in the virtual array file. This preallocates all blocks in the file. As

noted earlier, the contents of these blocks appended to the file are unknown. Use a FOR loop or the MAT ZER statement to initialize array values to a known (zero) quantity.

16.4.5 Simultaneous Access of a Virtual Array

The system gives write privileges only to the first program to open a file (array). When a second program attempts to modify an open array, the system reads the appropriate block from the disk but changes it only in the second program's buffer—not on the disk. When the second program references this array and attempts to read another block from the disk, a "?Protection violation" error occurs. This error occurs because the system attempts to update the disk with the information in the current block before the required block is read into memory.

Because the second program has no write privileges, it cannot update the disk. A CLOSE operation at this point also results in a "?Protection violation" error for the same reason. Once the job returns to BASIC–PLUS command level and you execute a NEW, OLD, or RUN command, the system performs a CLOSE on all channels. In this case, no write is attempted, so the CLOSE is successful.

To avoid simultaneous write accessing of a virtual array, determine whether your program has write privileges. Do this by testing the STATUS variable (see Section 17.1.1):

```
100      OPEN 'ARRAY' AS FILE #1%
110      IF (STATUS AND 1024%) &
            THEN PRINT 'NO WRITE ALLOWED ON ARRAY' &
            \    STOP
```

Do not use MODE 1% to open virtual array files. (MODE 1% allows you to open a file for simultaneous update by more than one program; see the *RSTS/E Programming Manual* for more information.) Your buffer is modified when you open an array with the MODE 1% option; thus, the system does not update the disk at this time.

Even when the first of several programs unlocks the file, allowing other programs to access the array, its modifications exist only in the first user's buffer. The system updates the array only when the first user accesses data from another block.

16.5 Programming Examples

As an example of virtual array usage, consider the problem of generating a large array of random numbers. Since a physical disk block is 256 words, an efficient array would contain a multiple of 256 elements. The following example uses a virtual array file, ARRAY1.DAT, that contains 5120 data elements in a 2–by–2560 array. Twenty physical blocks store this array. The program creates the virtual array V% by assigning a random value between 0 and 999 to each element in the array.

```
LISTNH
1000      DIM #3%, V%(1%,2559%)
1010      OPEN 'ARRAY1.DAT' AS FILE #3%
1020      FOR I% = 0% TO 1%
1030      V%(I%,J%) = RND * 1000% FOR J% = 0% TO 2559%
1040      NEXT I%
1050      CLOSE #3%
32767     END

Ready
```

After the file ARRAY1.DAT is created, you can access the virtual array elements by their subscripts. The next program prints every 256th value. Note that the array format in the DIM statement must be identical to the original format for predictable results. The file's channel number and the array's name can change, but the array must be formatted the same way every time it is accessed.

```
LISTNH
1000      DIM #4%, X%(1%,2559%)
1010      OPEN 'ARRAY1.DAT' AS FILE #4%
1020      FOR I% = 0% TO 1%
1030      PRINT X%(I%,J%); FOR J%=0% TO 2559% STEP 256%
1040      NEXT I%
1050      CLOSE #4%
1060      PRINT
32767     END

Ready

RUNNH
 204 909 954 839 65 131 537 784 371 798 565 173 122 910 39 8 318
  468 958 289

Ready
```

You can change values of array elements with the assignment statements LET, INPUT, or READ. This program changes the value of specified data elements:

```
LISTNH
1000 DIM #3%, V%(1%,2559%)
1010 OPEN 'ARRAY1.DAT' AS FILE #3%
1015 ON ERROR GOTO 1050
1020 INPUT 'ENTER THE I AND J LOCATION OF THE ELEMENT'; I%,J%
1030 N% = V%(I%,J%)
1040 INPUT 'ENTER THE NEW VALUE'; V%(I%,J%)
1045 PRINT &
     \ PRINT 'OLD VALUE WAS: ' ;N%, ' NEW VALUE IS: ';V%(I%,J%) &
     \ PRINT &
     \ GO TO 1020
1050 CLOSE #3%
32767 END

Ready
```

```
RUNNH
ENTER THE I AND J LOCATION OF THE ELEMENT? 0,9
ENTER THE NEW VALUE? 600

OLD VALUE WAS:  678   NEW VALUE IS:  600

ENTER THE I AND J LOCATION OF THE ELEMENT? 1,255
ENTER THE NEW VALUE? 333

OLD VALUE WAS:  937   NEW VALUE IS:  333

ENTER THE I AND J LOCATION OF THE ELEMENT? 0,2225
ENTER THE NEW VALUE? 9999

OLD VALUE WAS:  424   NEW VALUE IS:  9999

ENTER THE I AND J LOCATION OF THE ELEMENT? 9,9

Ready
```

The last two examples compare different access methods for virtual arrays. In the previous examples, ARRAY1.DAT was allocated as follows:

Block 1	V(0,0) – V(0,255)
Block 2	V(0,256) – V(0,511)
Block 3	V(0,512) – V(0,767)
.	.
.	.
.	.
Block 10	V(0,2304) – V(0,2559)
Block 11	V(1,0) – V(1,255)
.	.
.	.
.	.
Block 20	V(1,2304) – V(1,2559)

Notice that the second subscript varies from 0 to 2559 for each of the two values (0 and 1) of the first subscript. The system transfers a physical record (that is, a block) from the disk to memory at one time. Therefore, the system performs only one disk access for each group of 256 consecutive data elements; for example, V(0,256)–V(0,511). It is more efficient to sequentially access data elements in one block than to access data elements in different blocks.

The following two programs access, but do not print, each element in the virtual array. The first access method transfers a new block to memory for each data element accessed, resulting in 5120 disk accesses. The second method, however, transfers a new block to memory only once per 256 data elements, resulting in 20 disk accesses. The difference in execution time between both methods is significant.

Program 1
(Inefficient)

```
LISTNH
1000  DIM #3%, V%(1%,2559%)
1010  OPEN 'ARRAY1.DAT' AS FILE #3%
1015  T = TIME(0)
1020  FOR J% = 0% TO 2559%
1030  D% = V%(I%,J%) FOR I% = 0% TO 1%
1040  NEXT J%
1045  PRINT 'THIS ACCESS TOOK ';TIME(0) - T;'SECONDS.'
1050  CLOSE #3%
32767 END

Ready

RUNNH
THIS ACCESS TOOK  422 SECONDS.

Ready
```

Program 2
(Efficient)

```
LISTNH
1000  DIM #3%, V%(1%,2559%)
1000  OPEN 'ARRAY1.DAT' AS FILE #3%
1015  T = TIME(0)
1020  FOR I% = 0% TO 1%
1030  D% = V%(I%,J%) FOR J% = 0% TO 2559%
1040  NEXT I%
1045  PRINT 'THE SECOND ACCESS TOOK';TIME(0) - T ';SECONDS.'
1050  CLOSE #3%
32767 END

Ready

RUNNH
THE SECOND ACCESS TOOK 2 SECONDS.

Ready
```

Chapter 17
Block I/O

The I/O methods described so far are easy to use and flexible enough for many applications. But each method has its limits. Formatted ASCII I/O does not allow random access to data. In addition, formatted ASCII processing consumes CPU time as it scans for record delimiters and converts numeric data to and from ASCII format. Virtual arrays provide random access to files, but the files can be stored only on disks.

Block I/O, the third I/O method in BASIC–PLUS, does not have the limits of formatted ASCII or virtual arrays. Instead of performing I/O operations for you in a predefined way, BASIC–PLUS gives you access to the basic I/O services of the RSTS/E operating system. You can perform any type of physical I/O available under RSTS/E.

In block I/O, you control both the physical movement of data and the logical meaning of data. You can write programs that perform either or both types of operations using block I/O techniques.

In block I/O, your program directly controls the transfer of data between a device and its I/O buffer in your program area. You deal with a block I/O file as a series of physical records. Each physical record corresponds to one block on the storage device; the size of the block depends on the device you are using. In a disk file, each block has a number associated with it. This number gives you random access to each block in the file.

Because you deal with physical records in block I/O, you can tailor a program to the characteristics of a particular device. You can also take advantage of operating system features such as file sharing and multiterminal I/O. Most of the device-dependent programming techniques described in the *RSTS/E Programming Manual* are available to BASIC–PLUS programmers only through block I/O.

Besides controlling the transfer of data between the device and the I/O buffer, you also have to make data in the buffer meaningful to the rest of your program. In all types of BASIC–PLUS I/O, data in the I/O buffer is in unformatted binary form, just as it is on the storage device. In formatted ASCII and virtual array I/O, BASIC–PLUS automatically processes the data in the buffer and makes it available to your program as string, integer, or floating-point data. In block I/O, however, BASIC–PLUS does not process the data in the buffer. Instead, it provides you with a set of tools for processing the data yourself. You can read and write any file format using these tools.

Block I/O is the most powerful I/O method available in BASIC–PLUS. But it is also the most difficult to method to use because your program has to do most of the work that BASIC–PLUS does for you in formatted ASCII and virtual array I/O. You will need to learn several new programming techniques to use block I/O.

This chapter describes the statements and functions BASIC–PLUS provides for doing block I/O. The first part of the chapter describes the GET and PUT statements, which perform physical I/O. GET and PUT read and write blocks of data between a device and its I/O buffer. These statements have options that let you read and write specific blocks in a disk file.

The next part of the chapter describes the tools for making the data in the I/O buffer meaningful to your program:

FIELD statement	Lets you access the I/O buffer as string data by mapping string variables to buffer locations.
LSET and RSET statements	Move string values from your program into these buffer locations.
CVT conversion functions	Let you convert data from string to integer or floating-point format (and back) as you transfer it between the I/O buffer and your program.

This chapter also describes other BASIC–PLUS features that are often used with block I/O:

RECOUNT variable	Contains the number of characters read after an input operation.
STATUS variable	Contains information about the device you last opened.
BUFSIZ function	Returns the buffer size of an open channel.
SWAP% function	Swaps the low and high order bytes of an integer.

17.1 Opening a Block I/O File

As with all data files, you must open a file before executing block I/O statements on it. The format of an OPEN statement is:

OPEN <string> [{FOR INPUT / FOR OUTPUT}] AS FILE [#] <expression>

[,RECORDSIZE <exp>] [,CLUSTERSIZE <exp>] [,FILESIZE <exp>] [,MODE <exp>]

The OPEN statement opens an I/O channel and establishes an I/O buffer. See Chapter 14 of this manual for a description of the OPEN statement. The *RSTS/E Programming Manual* contains additional information about OPEN statement options. The rest of this section introduces the STATUS variable and the BUFSIZ function, which you use with the OPEN statement.

The STATUS variable and the BUFSIZ function return information about an open I/O channel. These tools are useful when you open a file without knowing anything about it. For example, if you are writing a general-purpose program, you may want to prompt the user for a file or device to open.

In this kind of program, you do not know in advance what the user will specify. You can use STATUS and BUFSIZ to get information about the file or device after you open it.

17.1.1 STATUS Variable

The STATUS variable contains information about the last channel on which your program executed an OPEN statement. The variable is a 16-bit word, each bit of which your program can test to determine status (see Section 11.7.4.1 for an example of bit testing). Table 17–1 shows the information, the tests, and the meaning of each bit. Note that the STATUS variable returns RSTS–specific information.

17.1.2 BUFSIZ Function

In certain applications, it is important for a program to determine the buffer size of an open channel. This is especially true if the OPEN statement specifies a logical device name. You can use the integer function BUFSIZ to extract this information.

The BUFSIZ function returns the size of the buffer for a specified open channel as an integer. For example:

```
20 Y%= BUFSIZ(N%)
```

This statement returns to Y% the size of the buffer in number of bytes for channel N%. If the channel is closed, the function returns 0.

Table 17–1: STATUS Variable

Bit	Test	Meaning
0–7	(STATUS AND 255%)	The first eight bits of the word contain the handler index. The following values apply for various devices. 0 Disk 2 Terminal 4 DECtape 6 Line Printer 8 Paper Tape Reader 10 Paper Tape Punch 12 Card Reader 14 Magnetic tape 16 PK: device (pseudo keyboard) 18 DX: device flexible diskette 20 RJ: device (2780 remote job entry) 22 NL: null device 24 DMC11/DMR11/ DDCMP Interface 26 Auto–Dialer 28 X–Y Plotter 30 TU58 DECtape II 32 KMC11 34 IBM Interconnect 38 DMP11/DMV11
8	(STATUS AND 256%)<>0%	The device is open for non-file-structured processing or is a non-file-structured device.
9	(STATUS AND 512%)<>0%	The job does not have read access to the device.
10	(STATUS AND 1024%)<>0%	The job does not have write access to the device.
11	(STATUS AND 2048%)<>0%	The device maintains its own horizontal position. Such devices are keyboards and line printers.
12	(STATUS AND 4096%)<>0%	The device accepts modifiers. Such devices use the record number as a modifier word rather than a physical position of the device. Keyboards, line printers, and card readers are such devices.
13	(STATUS AND 8192%)<>0%	Device is a character device.
14	(STATUS AND 16384%)<>0%	Device is an interactive device (keyboard).
15	(STATUS <0%)	Device is a random access blocked device, such as disk and non-file-structured DECtape.

17.2 Closing a Block I/O File

The CLOSE statement (described in Section 14.6) closes open I/O channels. It has the form:

CLOSE [#] <expression> [,[#] <expression> ,...]

The value of each expression specifies one of the 12 I/O channels.

Unlike formatted ASCII and virtual array I/O, a CLOSE statement on a block I/O file does not write the contents of the I/O buffer to the file before closing it. Instead, you must perform all I/O operations with block I/O files explicitly, using GET and PUT statements. Be sure that your program writes the last record to the file before you close it.

17.3 Reading and Writing Data – The GET and PUT Statements

You perform input and output operations with block I/O files directly between the device and the I/O buffer that the OPEN statement creates. Specify all I/O in terms of single records, using the GET and PUT statements. The formats for GET and PUT are:

```
GET #<exp>[,{BLOCK  <exp>} ,COUNT <exp>  ,USING <exp>]
           {RECORD <exp>}

PUT #<exp>[,{BLOCK  <exp>} ,COUNT <exp>  ,USING <exp>]
           {RECORD <exp>}
```

If you do not use the RECORD or BLOCK option, the GET statement reads the next sequential block from the file open on the channel designated by the first expression. The system places the block in the I/O buffer that is associated with the channel. The size of the block depends on the characteristics of the device that the file is on (see Table 17–2). In block I/O, the RECORD or BLOCK option refers to a sector whose length is device-specific, not to a logical data record.

When you use the RECORD or BLOCK options in a GET or PUT statement, the system accesses a specific block. For example:

```
100 GET #4%, RECORD 8%
```

This statement reads the eighth block of the file opened on channel 4 into your I/O buffer. Note that the preceding seven records of the file need not be read. This feature, not available in formatted ASCII files, is called random access.

Similarly, if you do not specify any of the options in the PUT statement, it writes the contents of the I/O buffer for the I/O channel onto the next sequential block of the file. The first expression specifies the channel number on which you opened the file.

The PUT statement writes a single block on the device. The exception to this is disk files. One PUT statement writes multiple disk blocks when you use the RECORDSIZE option in the OPEN statement to increase the I/O buffer size.

Table 17–2: Device Record Characteristics

Device	Record Characteristics
Disk	In file-structured processing, the default RECORDSIZE for disks is 512 bytes, the length of a disk block. In non-file-structured processing, however, the default RECORDSIZE depends on the device cluster size. (See the *RSTS/E Programming Manual* for information on non-file-structured processing.) You can specify a RECORDSIZE other than 512. When you do, though, keep this information in mind: 1. The RECORDSIZE must be an even number. 2. The GET and PUT statements transfer data in 512–byte units. Thus, you lose data when the RECORDSIZE is not a multiple of 512. In a GET, the system reads as many 512–byte units into the buffer as its size can accommodate and discards the portion of the last block that does not fit in the buffer. In a PUT, the system writes this last block as a partial block. The rest of this block has unpredictable contents. 3. The RECORDSIZE value changes the buffer size, but it does not change the record number associated with each block of the disk file. For example, when you specify a RECORDSIZE of 1024, the system transfers two disk blocks in each GET and PUT. However, each record in the the file still has the same number as it does when you use a 512–byte buffer. Use the BLOCK or RECORD option in GET and PUT statements to specify the correct record number for each read and write operation. (See Sections 17.3.1 and 17.3.2.)
DECtape	For file-structured DECtape, records are always 510 bytes long. For non-file-structured DECtape, records are always 512 characters.
Magnetic tape	When performing file-structured I/O, magnetic tape records are normally 512 characters. With non-file-structured I/O, magnetic tape records can be of any length. Only one record can be read per GET statement, and the record length cannot exceed the buffer size (as determined by the RECORDSIZE option). The minimum record size is 14 characters.
Keyboard	A keyboard record is a series of characters. The first delimiter marks the end of the record. The delimiter can be a RETURN, a LINE FEED, a FORM FEED, an ESCAPE, or a private delimiter (see the *RSTS/E Programming Manual*).
Card reader	A record consists of a single card. The RECORDSIZE option has no effect on card reader input.
Paper tape	RSTS/E reads a full buffer of input from the paper tape reader unless an end-of-tape is detected.

17.3.1 BLOCK Option

With disk files, you can perform random access I/O to any block of the file. Blocks in a disk file are always 512 bytes long and are logically numbered within the file from 1 to n, where n is the size of the file.

The BLOCK expression provides the logical block number of the file to the GET or PUT statement. The BLOCK option uses a real argument. This feature allows you to read and write specific blocks of files with more than 65535 blocks.

For example, assume that you open a disk file on channel 1. The following statement writes the contents of the I/O buffer associated with channel 1 on blocks 10 through 99 of that disk file:

```
200 PUT #1%, BLOCK I FOR I=10. TO 99.
```

You can read or write more than one physical block. Just assign a large I/O buffer to the file with the RECORDSIZE option in the OPEN statement. The size of the buffer does not affect the numbering of the blocks (512 bytes each) within the file.

If you open a disk file on channel 1 with a RECORDSIZE of 1024 (which causes two 512–byte blocks to be written with each PUT), you can write a PUT statement as:

```
200 PUT #1%, BLOCK I FOR I=10. TO 98. STEP 2.
```

When your program performs a random access GET or PUT on a disk file, the next GET or PUT statement on that channel accesses the next sequential block if no BLOCK number is specified. For example:

```
290 OPEN "DATA.DAT" AS FILE #1%, RECORDSIZE 512%
300 GET #1%, BLOCK 99.
310 PUT #1%
```

The PUT statement at line 310 writes block 100 of the disk file.

NOTE

Use of the BLOCK option in a GET statement is not compatible with other versions of BASIC.

17.3.2 RECORD Option

The RECORD option can also be used in GET and PUT statements to access a specific block. Unlike the BLOCK option, which uses a real argument, the RECORD option accepts an integer argument that specifies the logical block number in the file. Thus, the RECORD option limits the direct specification of a block number to 32767 (the largest integer value). You cannot use the RECORD option to work with large files. Note that the BLOCK and RECORD options are mutually exclusive.

17.3.3 COUNT Option

You can use the COUNT option in a PUT statement to specify the number of characters to write in the current record. The COUNT expression, however, cannot be greater than the size of the I/O buffer.

The COUNT option is usually used in non-file-structured processing. For example, if magnetic tape unit 0 is open for non-file-structured processing on channel 1, you can use the following statement to write an 80–character record:

```
100 PUT #1%, COUNT 80%
```

When you do not use COUNT, the PUT statement writes an entire buffer, regardless of whether the buffer contains data.

The COUNT option used in a GET statement specifies the maximum number of characters to be read in the current record regardless of the buffer size. For example, if magnetic tape unit 0 is open on channel 3, you can use the following statement to limit the GET operation to 50 characters:

```
250   GET #3%, COUNT 50%
```

When you do not use COUNT, the GET statement attempts to fill the entire buffer.

Note that the RECOUNT variable (see Section 17.3.5) is set in all cases. If the amount of data read by the GET operation is less than the limiting value of COUNT, the RECOUNT variable contains the actual amount of data read.

GET with the COUNT option works differently depending on the device type. For character-oriented devices (such as terminal, paper tape, or card reader), if the amount of data is greater than the limiting COUNT value, succeeding GET operations read the remaining data. For block-oriented devices (DECtape, magnetic tape, disk), the data beyond the limiting COUNT value in the last block is lost. Succeeding GET operations read from the next block.

NOTE

Use of the COUNT option in a GET statement is not compatible with other versions of BASIC.

17.3.4 USING Option

The RECORDSIZE option in the OPEN statement defines the size of an I/O buffer. The USING option can be used in a GET or PUT statement to specify an offset into that I/O buffer. That is, the USING option can cause a GET or PUT to begin a read or write operation at a specified byte in the buffer. For example:

```
200   GET #1%, COUNT 10%, USING 20%
```

Line 200 reads up to 10 characters into the I/O buffer associated with channel 1, beginning at a position 20 bytes into the buffer.

NOTE

Use of the USING option in a GET statement is not compatible with other versions of BASIC.

17.3.5 RECOUNT Variable

Non-file-structured devices can read less than a full buffer of data. To determine how much data was actually read, RECOUNT, a BASIC–PLUS variable, contains the number of characters read following every input operation. RECOUNT is used primarily for non-file-structured input; however, you can also use it with file-structured devices.

RECOUNT is set by every input operation on any channel (including channel 0). It is essential that you test or copy the RECOUNT value immediately after an INPUT or GET statement. The value of RECOUNT is not defined if an error occurs in the I/O operation. In addition, RECOUNT does not return useful information in immediate mode.

17.3.6 Extending Disk Files

When you create a disk file with an OPEN FOR OUTPUT (or OPEN) statement, the file has a length of 0. As blocks are written, the file grows in length. This growth is called *extending* the file.

A more exact description of the disk file extension process follows:

1. The system checks to see if there is room in the last cluster of a file for a new block. (The cluster size defines the minimum increment by which you can extend a file on the disk. A file need not occupy all blocks within the cluster.)
2. If so, then the file length is increased and previously unused space in that cluster is used.
3. If not, then a new cluster is appended to the file. Go back to Step 1.

The amount of space that the system allocates to a disk file may be greater than the file length. For example, if the file cluster size is four and you have written the first six blocks of that file, the file is six blocks long but has eight blocks (two clusters) of space.

You can extend a disk file by writing beyond the current end-of-file. However, a program must have write access to a file in order to extend it.

You can extend a file that is open in update mode, but only if you first lock block 1 of the file. You must make a contiguous file non-contiguous before you can extend it. (See the *RSTS/E Programming Manual* for more information about update mode and contiguous files.)

You can extend a file by a number of blocks at one time. For example:

```
100    OPEN "DATA.DAT" FOR OUTPUT AS FILE #1%
110    PUT #1%, BLOCK 100.
```

These statements create a file DATA.DAT and extend it immediately to 100 blocks. The system overhead for extending a file by a single block and by many blocks is nearly the same. Therefore, it is much more efficient to immediately extend a newly created file to its final length than to extend it many times in increments of a single block. Whenever you know the final size of a file, you should extend it to its full size in a single operation.

17.3.7 Alternate Buffer I/O

With normal I/O, moving data from one device to another requires two buffers, one for the input device and one for the output device. Your program also has to move data from the input buffer to the output buffer. But a special technique in BASIC–PLUS allows you to bypass one buffer. This technique is called "alternate buffer I/O."

With alternate buffer I/O, you can:

- Move data from an input buffer directly to an output device, without using an output buffer.
- Move data from an input device directly to an output buffer, without using an input buffer.
- Move data between input or output devices and a temporary buffer set up as a work area.

These techniques are useful for copying one file to another and for record blocking and deblocking. (You block and deblock records with the FIELD statement, which is described in Section 17.4.1. For an example of record blocking and deblocking, see Section 17.6.)

To perform alternate buffer I/O, you replace the channel number in a GET or PUT statement with an expression of the form:

```
SWAP%(B%) + I%
```

B% is the channel number of the buffer to be used, and I% is the channel number on which I/O occurs.

The following example shows a fast copy, using alternate buffer I/O:

```
05    ON ERROR GO TO 9000
10    OPEN "RANDOM" FOR INPUT AS FILE #1%
20    OPEN "COPY" FOR OUTPUT AS FILE #2%, RECORDSIZE 32767%+1%+2%
30    GET #1%
40    PUT #SWAP%(1%) + 2%
50    GO TO 30
60    PRINT "FILE COPIED"
70    CLOSE #1%, #2%
80    STOP
9000  IF ERR=11% AND ERL=30% THEN &
      RESUME 60 &
      ELSE ON ERROR GO TO 0
32767 END
```

In this example, the input device is open on channel 1 and the output device is open on channel 2. The GET statement in line 30 moves data from the input device into the input buffer. The PUT statement in line 40 moves the contents of the input buffer directly to the output device open on channel 2. With normal I/O, you have to write two FIELD statements, one for each buffer, and use an LSET statement to move the data from one buffer to another (see Section 17.4).

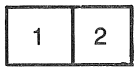
You can also use alternate buffer I/O as a spacesaving technique in programs that transfer data among several different files. Open each file with a small buffer size (for example, 2 bytes) because you will not be using these I/O buffers for data transfers. Open the null device (NL:) to create a "scratch buffer" at the size you need for data transfers. (You have to use the RECORDSIZE option when you open the null device because its default buffer size is 2 bytes.) You can then use alternate buffer I/O to transfer data from an input file to the scratch buffer and from the scratch buffer to an output file.

You use the expression SWAP%(B%) + I% to specify alternate buffer I/O because BASIC–PLUS stores information about an I/O channel in one word of memory. The low byte of the word contains the channel number; the high byte of the word contains the channel number of the buffer to use for I/O on that channel. When you specify only a channel number, the high byte of this word is 0. BASIC–PLUS uses the value in the low byte as both the buffer number and the channel number.

The expression SWAP%(B%) + I% creates a one-word value that contains a nonzero buffer number in the high byte and a channel number in the low byte. For example:

```
SWAP%(1%) + 2%
```

This expression combines the integers 1% and 2% into a value with the internal format:



High Byte Low Byte

When you use this expression in a GET or PUT statement, BASIC–PLUS tells the system to transfer data to or from the buffer specified in the high byte instead of the I/O channel's default buffer. (See Section 17.5.2 for more information about SWAP%.)

17.4 Accessing the I/O Buffer

So far this chapter has described how to read and write data between a device and an I/O buffer with GET and PUT statements. This section describes the statements you use to make data in an I/O buffer available to your program.

BASIC–PLUS has three statements that operate on data in an I/O buffer: FIELD, LSET, and RSET. FIELD lets you access the buffer as string data, and LSET and RSET move string values from your program into the buffer. With these statements, you can access and modify each byte of an I/O buffer.

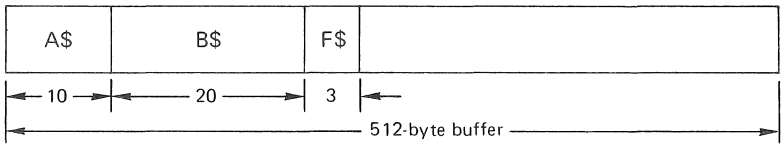
17.4.1 FIELD Statement

The FIELD statement dynamically associates string variables with parts of an I/O buffer. The FIELD statement has the form:

FIELD #<exp>,<exp1> AS <string var1>[,<exp2> AS <string var2>,...]

The <exp> is a channel number associated with some file by an OPEN statement, <exp1> is the length in characters of the associated string variable, and <string var1> is a string variable name. The names are associated from left to right with successive characters in the I/O buffer assigned to the designated channel number. For example:

```
75 FIELD #2%, 10% AS A$, 20% AS B$, 3% AS F$
```



As shown in the previous diagram, statement 75 associates three strings, A$, B$, and F$ in the I/O buffer, with lengths of 10, 20, and 3 characters, respectively. This statement represents a total number of 33 characters. The total number of characters must be less than or equal to the actual I/O buffer size (this depends on the device and the RECORDSIZE option; see Section 14.3.1).

FIELD statements do not move data. Instead, they permit direct access to sections of the I/O buffer through string variables. The effect on a string variable is temporary and is nullified by any attempt to assign a value to the variable (other than with the LSET and RSET statements; see Section 17.4.3). For example:

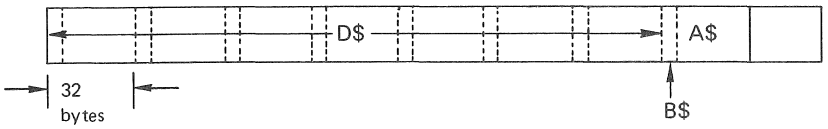
```
100    OPEN 'FILE' AS FILE #2%
110    FIELD #2%, 5% AS A$
120    LET A$ = 'ABCDE'
```

The LET statement at line 120 removes the string variable A$ from the I/O buffer.

A FIELD statement is an executable statement. You can change a buffer description at any time by executing another FIELD statement. For example, suppose that each block of a disk file contains sixteen 32–character

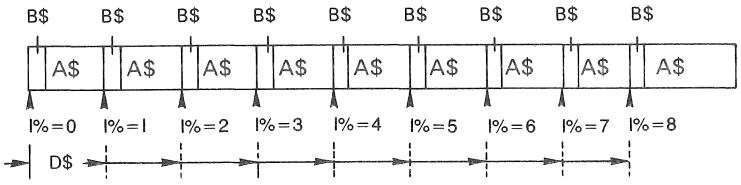
subrecords. Each subrecord consists of one 5–character field and one 27–character field. To extract the eighth subrecord from the I/O buffer, you can execute the following statement:

```
200 FIELD #1%, 224% AS D$, 5% AS B$, 27% AS A$
```



Line 200 causes the string variables B$ and A$ to point to the desired subrecord. The string D$ is created to permit the first seven subrecords (7*32=224) to be skipped. You can use an even more general statement to obtain any of the subrecords in the I/O buffer:

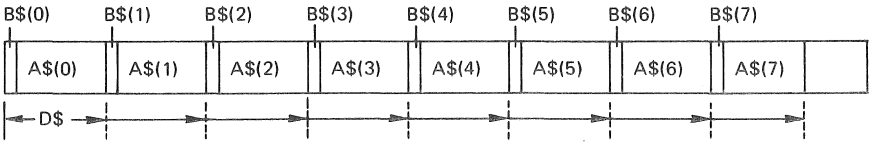
```
190       FOR I% = 0% TO 15%
200       FIELD #1%, I%*32% AS D$, 5% AS B$, 27% AS A$
210       NEXT I%
```



This FOR loop shows how you can repetitively execute a FIELD statement. Each time the FIELD statement is executed, I% contains the number of the subrecord that B$ and A$ are to contain, as an integer from 0 to 15. When I%=0, for example, the expression I%*32% equals 0. B$ points to the first subrecord in the buffer. When I%=1, however, the expression I%*32% equals 32, so B$ now points to the first subrecord beyond the 32nd character of the buffer. Each single increment of I% moves B$ 32 characters further into the buffer.

You can also use subscripted string variables in FIELD statements. For example, the following statements allocate the subrecords described in the previous example to two string arrays:

```
300       DIM A$(15%), B$(15%)
310       FOR I% = 0% TO 15%
320       FIELD #1%, I%*32% AS D$, 5% AS B$(I%), 27% AS A$(I%)
330       NEXT I%
```



With each iteration of the FIELD statement at line 320, the dummy string D$ increases by 32 characters. This makes the displacement from the start of I/O buffer to the string B$(I%) equal to 32 times I% characters. Once the system executes this loop, it fixes the position of each string in the arrays A$ and B$. A$(0) and B$(0) point to the first subrecord, and A$(15) and B$(15) point to the last.

You can also use more than one FIELD statement in a program to define the same buffer in different ways. For example:

```
20 FIELD #1%, 40% AS WHOLE.FIELD$ &
   \ FIELD #1%, 10% AS A$, 10% AS B$, 10% AS C$
```

The first FIELD statement associates the first 40 characters of a buffer with the variable WHOLE.FIELD$. The second FIELD statement associates the first 30 characters of the same buffer with the variables A$, B$, and C$. Later program statements can refer to any of these variables. (The first example in Section 17.6 illustrates this use of multiple FIELD statements.)

You must not define virtual array strings as string variables in a FIELD statement. When you define strings as virtual arrays, they must be in a fixed place, in both a disk file and the I/O buffer, for that file. Attempting to specify a virtual array string variable in a FIELD statement has no effect on the virtual array string.

17.4.2 The LSET and RSET Statements

After you define string variables as part of the I/O buffer with a FIELD statement, you can store values in these variables without moving them from the I/O buffer. The LSET and RSET statements store values in a string variable without redefining the string position. These statements have the form:

LSET <string var> [,<string var>,...] = <string>

RSET <string var> [,<string var>,...] = <string>

Here <string var> is any legal existing string variable name. You can separate multiple string variable names by commas. The <string> is any legal string expression.

The LSET and RSET statements store the value of the string expression into the designated string or strings. The string previously stored in the variable is overwritten, although the length of the string is not changed.

If a new string is longer than an existing string, the system truncates the new string. If a new string is shorter than an existing string, it is either padded with spaces on the right by LSET or padded with spaces on the left with RSET. LSET causes the string to be left-justified in the field, and RSET causes the string to be right-justified.

LSET and RSET normally move data to an I/O buffer, as this section describes. They can also move a value into any string variable in your program.

17.4.3 Differences between the LET Statement and the LSET/RSET Statements

The LET statement cannot place string values into an I/O buffer because it redefines the string variable. Another restriction on LET occurs when a statement equates two strings:

```
100 LET A$ = B$
```

To avoid unnecessary character manipulation, BASIC–PLUS makes A$ and B$ to point to the same string in memory. Normally, any operation that alters B$ causes that string to be moved, so that no conflict arises. However, LSET and RSET do not move strings; they change existing strings in a fixed position.

If you change the value of B$ in line 100 with an LSET or RSET statement, then you also change the value of A$. For example:

```
LISTNH
400 B$ = "ABC"
410 A$ = B$
420 LSET B$ = "XYZ"
430 PRINT A$
32767 END

Ready

RUNNH
XYZ

Ready
```

Both A$ and B$ contain "XYZ" after the execution of line 420.

Note that if you define the string B$ in this example with a FIELD statement as pointing to a subrecord in some I/O buffer, the string A$ also points to the same I/O buffer (being identical to B$). Executing a GET statement to read another record into the I/O buffer then changes the value of A$ as well as B$. For this reason, you should use LSET and RSET only for block I/O operations; using these statements for other purposes may cause unexpected results.

When the strings A$ and B$ should not be physically identical, use the following method to move the string B$ into the string A$:

```
300 LET A$ = B$ + ""
```

This statement causes BASIC–PLUS to create a new string for A$, rather than pointing A$ and B$ to the same string.

17.5 Converting Numeric Data – CVT Functions and SWAP%

This section describes functions that:

- Convert data between numeric and string formats
- Swap the high and low order bytes of an integer

Both types of functions are useful in block I/O.

17.5.1 CVT Conversion Functions

The BASIC–PLUS FIELD, LSET, and RSET statements let you access data in the I/O buffer as string data only. To process numeric data in your program, you must convert it from string to numeric format after you move it from the I/O buffer. After processing, you must convert the numeric data back to string data before you transfer it to the I/O buffer for output.

BASIC–PLUS provides four CVT conversion functions to convert data between string format and integer or floating-point formats. Table 17–3 lists these four conversion functions. (A fifth CVT function, CVT$$, helps you process string data. See Chapter 10.)

NOTE

Do not confuse the CVT conversion functions with the CHANGE statement or the NUM$ and VAL functions.

Table 17–3: CVT Conversion Functions

Function Form	Operation
A$ = CVT%$(I%)	Maps an integer into a two-character string.
I% = CVT$%(A$)	Maps the first two characters of a string into an integer. If the string has fewer than two characters, BASIC–PLUS appends null characters as required.
A$ = CVTF$ (X)	Maps a floating-point number into a four- or eight-character string (depending upon whether you are using the two-word or four-word math package on the system). You can determine the current math package by examining LEN(CVTF$(0)).
X = CVT$F (A$)	Maps the first four or eight characters (depending upon whether you are using the two-word or four-word math package on the system) of a string into a floating-point number. If the string has fewer than the required number of characters, BASIC–PLUS appends null characters.

These CVT functions affect the value's storage format, but not its value. Each character in a string requires one byte of storage (8 bits). Hence characters can assume (decimal) values from 0 through 255 only. A 16–bit

quantity can be defined as either an integer or a two-character string. You can similarly define two-word floating-point numbers as four-character strings and four-word floating-point numbers as eight-character strings.

The CVT functions that change storage format perform two important functions:

- They permit dense packing of data in records. For example, you can pack any integer value between –32768 and 32767 in a record in two characters using CVT%$. This is true only for integers between –9 and 99 when data is stored as ASCII characters.
- Converting the internal numeric representation to an ASCII string (as with the NUM$ function) is a more time-consuming process than that performed by the CVT functions. Thus, the CVT functions speed the processing of a large amount of data in a file.

Use the CVT functions in LSET, RSET, and LET statements to transfer numeric data between an I/O buffer and your program. For example, the following statements store two integers, U% and CL%, and two single-precision floating-point numbers, X and Y, in the I/O buffer for channel 8:

```
200 FIELD #8%, 2% AS U$, 2% AS CL$, 4% AS X$, 4% AS Y$
210 LSET U$ = CVT%$(U%) &
    \ LSET CL$ = CVT%$(CL%) &
    \ LSET X$ = CVTF$(X) &
    \ LSET Y$ = CVTF$(Y)
```

You can retrieve this data from the buffer by converting it from string to numeric form:

```
220 LET U% = CVT$%(U$) &
    \ LET CL% = CVT$%(CL$) &
    \ LET X = CVT$F(X$) &
    \ LET Y = CVT$F(Y$)
```

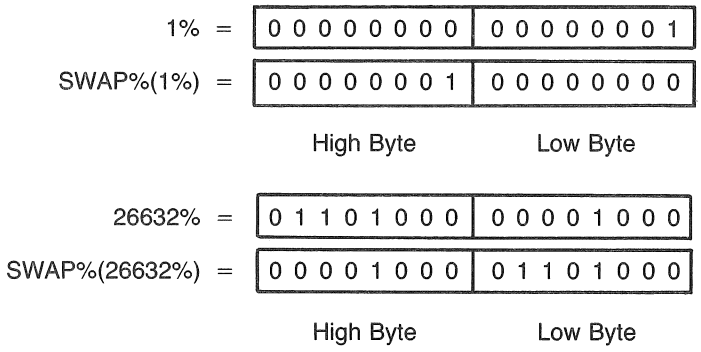
Use LSET or RSET statements to move data from your program into the I/O buffer; use LET statements (or other statements that assign values to variables) to move data from the I/O buffer to your program's data area.

17.5.2 SWAP% Function

The SWAP% function swaps the low order byte of an integer with its high order byte. SWAP% returns an integer value with the bytes reversed. For example:

```
PRINT SWAP%(1%)
 256

PRINT SWAP%(26632%)
 2152
```

The following diagrams show the internal formats of the integers 1% and 26632% before and after using SWAP%:



SWAP% has several different uses in BASIC–PLUS programming. For example, Section 17.3.7 shows the use of SWAP% in alternate buffer I/O to create a one-word value that contains a channel number in the low byte and a nonzero buffer number in the high byte. SWAP% is useful for reading data returned by system function calls. See Section 11.7.4.3 for an example and the *RSTS/E Programming Manual* for details. You also need to use SWAP% with the CVT functions when you read numeric data using block I/O that was written using a different I/O method. See Section F.4 for more information.

17.6 Block I/O Examples

The examples in this section show various ways to use block I/O. Some examples are complete programs, while others are program fragments. These examples are intended as an introduction — block I/O has many more possible uses.

The following example prints a line on the terminal:

```
5  EXTEND
10 OPEN 'KB:' FOR OUTPUT AS FILE #1% !Create 128-byte buffer
20 FIELD #1%, 40% AS WHOLE.FIELD$    !Define entire buffer &
                                                          &
   \ FIELD #1%, 10% AS A$, 10% AS B$, 10% AS C$ &
   !Then define three subfields in same buffer &

30 LSET WHOLE.FIELD$ = SPACE$(40%)  !Preload buffer with blanks
40 LSET A$ = "12345"                !Load first 10 characters &
   \ RSET B$ = "67890"              !Load next 10 characters &
   \ RSET C$ = "VWXYZ"              !Load last 10 characters
50 PUT #1%, COUNT 40%               !Write this buffer
60 PRINT #1%, '**'                  !Show that PUT gives no <CR><LF>
32767 END
```

When you run this program, it prints:

```
12345          67890     VWXYZ          **

Ready
```

In this example, the OPEN statement at line 10 opens KB: with the default buffer size (128 bytes). The FIELD statements at line 20 define the buffer, once as a 40–character string WHOLE.FIELD$ and once as three 10–character strings A$, B$, and C$.

The LSET statement at line 30 preloads WHOLE.FIELD$ (the first 40 characters of the buffer) with blanks to delete whatever data is in that part of the buffer. (In this case, the blanks delete the program header line that BASIC–PLUS prints when you run the program. RUNNH stops the header from being printed, but it is still in the buffer.)

The LSET and RSET statements at line 40 use the variables A$, B$, and C$ to load the first 30 characters of the buffer with the line to be printed. (Note that both LSET and RSET write blanks into unused portions of the 10–character fields.) The PUT statement prints the line, and the PRINT statement shows that the PUT statement does not output a carriage return/line feed.

The next example is a program to move data in a file named REPORT.FIL from a disk to the line printer:

```
50 EXTEND
110 OPEN 'REPORT.FIL' AS FILE #1%
120 ON ERROR GOTO 200
130 OPEN 'LP:' FOR OUTPUT AS FILE #2%, RECORDSIZE 512%
140 FIELD #1%, 512% AS A$
150 FIELD #2%, 512% AS B$
160 GET #1%
170 LSET B$ = A$
180 PUT #2%
190 GOTO 160
200 CLOSE #1%, #2%
32767 END
```

In this example, the buffers for the line printer and the disk file are both initialized to 512 characters. The FIELD statements at lines 140 and 150 define A$ and B$ to refer to these buffers.

The LSET statement transfers data read at line 160 to the line printer buffer. (RSET can also be used here because A$ and B$ are the same length.) The PUT statement at line 180 outputs the data to the line printer. The loop terminates when the "?End of file on device" error occurs. The ON ERROR GOTO statement transfers control to line 200, which closes both I/O channels.

Note that you can write a more efficient version of this program using alternate buffer I/O, which is described in Section 17.3.7.

The next two examples show how you can use CVT functions to store numeric data in compact internal form. The first program creates a magnetic tape file that has 26 records in each block. Each record consists of an integer in two-byte internal format and a one-byte string. The second program reads the file and prints the data on the terminal.

```
50  EXTEND
100 DIM INTEGER$(25%), LETTER$(25%) !Define arrays of string &
                                    !variable names
200 OPEN 'MM0:CVT.BLK' FOR OUTPUT AS FILE #1%, RECORDSIZE 78% &

300 FIELD #1%, I%*3% AS D$,         !Offset over previous records &
        2% AS INTEGER$(I%),         !Storage for converted integer &
        1% AS LETTER$(I%)           !Storage for 1-character string &
    FOR I% = 0% TO 25%              !FOR modifier defines record 26 times &

400 FOR I% =0% TO 25%               !Now build one physical block
410 LET J% = I% + 65%
500 LSET INTEGER$(I%) = CVT%$(J%)   !Store "internal" integer
600 LSET LETTER$(I%) = CHR$(J%)     !Store corresponding letter
700 NEXT I%
800 PUT #1%                         !Write the record onto tape
32767 END


50  EXTEND
100 DIM INTEGER$(25%), LETTER$(25%), INTEGER%(25%)
200 OPEN 'MM0:CVT.BLK' FOR INPUT AS FILE #2%, RECORDSIZE 78%
210 GET #2%
300 FIELD #2%, I%*3% AS D$,         !Offset over previous records &
               2% AS INTEGER$(I%),  !Converted integer &
               1% AS LETTER$(I%)    !1-character string &
    FOR I% = 0% TO 25%              !FOR modifier defines record 26 times &

400 FOR I% =0% TO 25%
500 LET INTEGER%(I%) = CVT$%(INTEGER$(I%)) !Convert stored integer back &
                                           !to internal format
600 PRINT INTEGER%(I%), LETTER$(I%)        !Print each integer and letter
700 NEXT I%
32767 END
```

The next example shows how you can use FIELD statements to block and deblock records. "Deblocking" a record refers to the process of breaking a physical record from an input device into the logical components that you process in your program. "Blocking" refers to the process of converting these logical components back to their physical form for output. In formatted ASCII and virtual array I/O, BASIC–PLUS handles blocking and deblocking for you. In block I/O, you block and deblock records by manipulating data in a buffer with FIELD, LSET, and RSET statements.

```
100 GET #2%                         !Read a block
200 FOR I% = 0% TO 420% STEP 80%    !Process each record in block
300     FIELD #2%, I% AS D$,        !D$ is not used, it's just an offset &
                   80% AS R$        !This is the current record &
          .                         !Process the current record &
!         .                                                     &
!         .
400 NEXT I%                         !Field next record in block
500 PUT #2%                         !Rewrite modified block
```

The following program fragment shows how to use the null device and FIELD to set up a buffer to use for breaking down an arbitrary string:

```
50  EXTEND
300 OPEN 'NL:' AS FILE #12%, RECORDSIZE 132%   !Create a dummy buffer &
                                               !(Won't be read or written)
400 FIELD #12%, 132% AS WHOLE.LINE$,           !Give a name to whole buffer &
    \ FIELD #12%, 10% AS ITEM$,                !Create subfields in buffer &
                      4% AS D$,                !Won't reference D$ fields &
                     50% AS DESCRIPTION$,                                  &
                      6% AS D$,                                            &
                     16% AS QUANTITY$,                                     &
                      5% AS D$,                                            &
                     18% AS UNIT.PRICE$,                                   &
                      5% AS D$,                                            &
                     18% AS TTL.PRICE$,                                    &
                      2% AS D$                                             &
!   ,                                                                       &
!   ,                                                                       &
!   ,
500 INPUT LINE #L%, D$                    !Read a record from input file    &
    \ LSET WHOLE.LINE$ = L$               !Move it into this buffer         &
    \ QUANTITY = VAL(QUANTITY$)           !Change strings into numbers      &
    \ UNIT.PRICE = VAL(UNIT.PRICE$)       !where appropriate                &
    \ EXTENDED.PRICE = VAL(EXTENDED.PRICE$)
```

17.7 UNLOCK Statement

Block I/O gives you access to a RSTS/E feature called file sharing. To use this feature, you open a file in update mode (MODE 1%). Update mode gives multiple users write access to a file. At the same time, it prevents simultaneous writing of the same data. When a program performs a read operation on a file open in update mode, the system "locks" the blocks that are accessed. This "lock," called an implicit lock because the system does it automatically, ensures that no other user can modify the blocks.

You can release an implicit lock with the UNLOCK statement. It has the form:

UNLOCK #<expression>

The <expression> is the channel number of the file that is open for update.

You can explicitly lock and unlock blocks in a file with the SPEC% function. See the *RSTS/E Programming Manual* for more information about update mode and the SPEC% function.

APPENDIXES

Appendix A
Language Summary

A.1 Summary of Variable Types

In EXTEND mode, a variable name can consist of a letter, followed by 0 to 29 additional characters, each a letter, digit or a period.

Examples

`PER.DIEM.FACTOR`	(floating-point variable)
`Z%`	(integer valid also in NOEXTEND)
`BRANCH.CONTROL%`	(integer variable)
`HEADING.A31.FORM$`	(string variable)
`DECK.OF.CARDS$(3,12)`	(string array)

In NOEXTEND mode a variable name consists of a letter optionally followed by a digit. The rules for specifying integers, strings, and dimension elements are the same in EXTEND and NOEXTEND modes.

Type	Variable Name	Examples

Floating-Point	A variable name with no suffix	`A`
		`I`
		`X3`

Integer	Any variable name followed by a % character	`B%` `D7%`
Character String	Any variable name followed by a $ character	`M$` `R1$`
Floating-Point Array	Any floating-point variable name followed by one or two dimension elements in parentheses	`S(4)` `E(5,1)` `N2(8)` `V8(3,3)`
Integer Array	Any integer variable name followed by one or two dimension elements in parentheses	`A%(2)` `I%(3,5)` `E3%(4)` `R2%(2,1)`
Character String Array	Any string variable name followed by one or two dimension elements in parentheses	`C$(1)` `S$(8,5)` `A2$(8)` `V1$(4,2)`

A.2 Summary of Operators

Type		Operator	Operates On
Arithmetic	–	Unary minus	Numeric variables and constants.
	^ or **	Exponentiation	
	*, /	Multiplication, division	
	+, –	Addition, subtraction	
Relational	=	Equals	String or numeric variables and constants.
	<	Less than	
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
	<>	Not equal to	
	==	Approximately equal to (numbers); Identically equal to (strings)	
Logical	NOT	Logical negation	Relational expressions composed of string or numeric elements, integer variables, or integer expressions.
	AND	Logical product	
	OR	Logical sum	
	XOR	Logical exclusive or	
	IMP	Logical implication	
	EQV	Logical equivalence	
String	+	Concatenation	String constants and variables.
Matrix	+, –	Addition and subtraction of matrices of equal dimensions, one operator per statement	Subscripted variables.
	*	Multiplication of conformable matrices	
	*	Scalar multiplication of a matrix	

A.3 Summary of Functions and Variables

This summary includes BASIC–PLUS standard functions, optional functions, and built-in variables (such as ERL).

Each function or variable has the format:

Y = function

or

Y = variable

Y is the value returned. Y with no suffix means that the function or variable returns a floating-point value. A % suffix means that the function or variable returns an integer value; a $ suffix means that the function or variable returns a string value.

Function arguments can be floating-point, integer, or string. A function argument without a suffix is a floating-point argument; a % or $ suffix indicates an integer or string argument.

You can always replace a floating-point argument with an integer argument. You can also replace an integer argument with a floating-point argument (an implied FIX is done) except in the CVT%$ and MAGTAPE functions. (The I% argument in these functions means you must supply an integer.) However, for the greatest efficiency, use an argument of the type shown.

Type	Function	Description
Mathematical Functions (Optional)	Y=ABS(X)	Returns the absolute value of X.
	Y=ATN(X)	Returns the arc tangent (in radians) of X.
	Y=COS(X)	Returns the cosine of X, where X is in radians.
	Y=EXP(X)	Returns the value of e^X, where e=2.71828...
	Y=FIX(X)	Returns the truncated value of X, SGN(X)*INT(ABS(X)).
	Y=INT(X)	Returns the greatest integer in X that is less than or equal to X.
	Y=LOG(X)	Returns the natural logarithm of X, log(e)X.
	Y=LOG10(X)	Returns the common logarithm of X, log(10)X.
	Y=PI	Returns the constant 3.14159... .
	Y=RND	Returns a random number between 0 and 1.
	Y=RND(X)	Returns a random number between 0 and 1.
	Y=SGN(X)	Returns the sign function of X; +1 if positive, 0 if zero, –1 if negative.
	Y=SIN(X)	Returns the sine of X, where X is in radians.
	Y=SQR(X)	Returns the square root of X.
	Y=TAN(X)	Returns the tangent of X, where X is in radians.
Print Functions	Y%=CCPOS(X%) or Y%=POS(X%)	Returns the current position of the print head for I/O channel X%; channel 0% is your terminal.
	Y$=TAB(X%)	Moves print head to position X% in the current print record, or is disregarded if the current position is beyond X%. (The first position is counted as 0.)
String Functions	Y%=ASCII(A$)	Returns the ASCII value of the first character in the string A$.
	Y$=CHR$(X%)	Returns the character that has the ASCII value of X. Only one character is generated.

(continued on next page)

Type	Function	Description
String Functions	Y$ = CVT%$(I%)	Maps an integer into a two-character string.
	Y$ = CVTF$(X)	Maps a floating-point number into a four- or eight-character string.
	Y% = CVT$%(A$)	Maps the first two characters of string A$ into an integer.
	Y = CVT$F(A$)	Maps the first four or eight characters of string A$ into a floating-point number.
	Y$ = CVT$$(A$,I%)	Converts string A$ to string Y$ according to the value of I%.
	Y$ = RAD$(N%)	Converts an integer value to a three-character string; is used to convert from Radix–50 format to ASCII (see the *RSTS/E Programming Manual*).
	Y% = SWAP%(N%)	Causes a byte swap operation on the two bytes in the integer variable N%.
	Y$ = STRING$(N1%,N2%)	Returns string Y$ of length N1%, composed of characters whose ASCII decimal value is N2%.
	Y$ = LEFT(A$,N%)	Returns a substring of the string A$ from the first character to the N%th character (the leftmost N% characters).
	Y$ = RIGHT(A$,N%)	Returns a substring of the string A$ from the N%th to the last character (the rightmost characters of the string starting with the N%th character).
	Y$ = MID(A$,N1%,N2%)	Returns a substring of the string A$ that is N2% characters long and starts with character N1% (the characters between and including the N1% to N1% + N2%–1% characters).
	Y% = LEN(A$)	Returns the number of characters in the string A$, including trailing blanks.
	Y% = INSTR(N1%,A$,B$)	Indicates a search for the substring B$ within the string A$ beginning at character position N1%. Returns a value 0 if B$ is not in A$. Returns the character position of B$ if B$ is in A$ (character position is measured from the start of the string).
	Y$ = SPACE$(N%)	Returns a string of N% spaces; is used to insert spaces within a character string.

(continued on next page)

Type	Function	Description
String Functions	Y$ = NUM$(N%)	Returns a string of numeric characters representing the value of N% as it is output by a PRINT statement. For example: NUM$(1.0000) = (space)1(space) NUM$(–1.0000) = –1(space).
	Y$ = NUM1$(N)	Returns a string of characters representing the value of N%. NUM1$ is similar to the function NUM$, except that it does not return spaces or E-format results.
	Y = VAL(A$)	Computes the numeric value of the string of numeric characters A$. For example: `VAL("15")=15` If A$ contains any character not acceptable as numeric input with the INPUT statement, an error results.
	Y$ = XLATE(A$,B$)	Translates A$ to the new string Y$ using the table string B$.
String Arithmetic Functions (Optional)	Y$ = SUM$(A$,B$)	Returns a numeric string equal to the arithmetic sum of numeric strings A$ and B$.
	Y$ = DIF$(A$,B$)	Returns a numeric string equal to the arithmetic difference A$–B$ of numeric strings A$ and B$.
	Y$ = PROD$(A$,B$,P%)	Returns a numeric string equal to the product of numeric strings A$ and B$, rounded or truncated to P% places.
	Y$ = QUO$(A$,B$,P%)	Returns a numeric string equal to the arithmetic quotient A$/B$ of numeric strings A$ and B$, rounded or truncated to P% places.
	Y$ = PLACE$(A$,P%)	Returns a numeric string equal to the numeric string A$, rounded or truncated to P% places.
	T% = COMP%(A$,B$)	Returns a value reflecting the result of an arithmetic comparison between numeric strings A$ and B$. T% = –1 for A$<B$, 0 for A$ = B$, and 1 for A$>B$.

System Functions and Variables	Y$ = DATE$(0%)	Returns the current date in the format:
		`02-Mar-71`
		or
		`71,03,02`

(continued on next page)

Type	Function	Description
System Functions and Variables	Y$ = DATE$(N%)	Returns a character string corresponding to a calendar date: N% = (day of year) + [(number of years since 1970)*1000] DATE$(1%) = 01-Jan-70 or 70,01,01
	Y$ = TIME$(0%)	Returns the current time of day as a character string: TIME$(0%) = 05:30 PM or 17:30
	Y$ = TIME$(N%)	Returns a string corresponding to the time at N% minutes before midnight. For example: TIME$(1%) = 11:59 PM or 23:59 TIME$(1440%) = 12:00 AM or 00:00 TIME$(721%) = 11:59 AM or 11:59
	Y = TIME(0%)	Returns the clock time in seconds since midnight as a floating-point number.
	Y = TIME(1%)	Returns the central processor time used by the current job in tenths of seconds.
	Y = TIME(2%)	Returns the connect time (during which you are logged into the system) for the current job in minutes.
	Y = TIME(3%)	Returns the decimal number of kilocore ticks (KCTs) that this job uses.
	Y = TIME(4%)	Returns the decimal number of minutes of device time that this job uses.
	Y% = STATUS	Returns the status of the I/O channel most recently opened.
	Y% = BUFSIZ(N)	Returns the buffer size of the device or file open on channel N.
	Y% = LINE	Returns the line number of the statement being executed at the time of a CTRL/C interrupt.
	Y% = ERR	Returns the value associated with the last encountered error if an ON ERROR GOTO statement appears in the program.
	Y% = ERL	Returns the line number at which the last error occurred if an ON ERROR GOTO statement appears in the program.

(continued on next page)

Type	Function	Description
Matrix Functions and Variables (Optional)	MAT Y=TRN(X)	Returns the transpose of the matrix X.
	MAT Y=INV(X)	Returns the inverse of the matrix X.
	Y=DET	Following an INV(X) function evaluation, the variable DET is equivalent to the determinant of X.
	Y%=NUM	Following input of a matrix, NUM contains the number of rows input or, in the case of a one-dimensional matrix, the number of elements entered.
	Y%=NUM2	Following input of a matrix, NUM2 contains the number of elements entered in that row.
Input/Output	Y%=RECOUNT	Returns the number of characters read following an input operation. Used primarily with non-file-structured devices.

A.4 Summary of Statements

The following summary lists in alphabetical order the general formats of BASIC–PLUS language statements and statement modifiers. For more detailed information, refer to the section or sections in the manual that describe the statement.

Statement elements and their abbreviations are:

variable or var	Any legal variable (see Section 8.3.2 or A.1).
line number	Any legal line number (see Section 7.2.1).
expression or exp	Any legal expression (see Section 8.4).
message	Any combination of characters.
condition or cond	Any relational or logical expression (see Sections 8.4.3 and 8.4.4).
constant	Any integer constant (see Section 8.3.1). You do not have to include a % character.
argument(s) or arg	Dummy variable names.
statement	Any legal BASIC–PLUS statement.
string	Any legal string constant, variable, function, or expression.
protection	Any legal protection code, as described in the *RSTS/E System User's Guide*.

value(s)	Any floating-point, integer, or string constant (see Section 8.3.1).
list	The legal list for the statement.
dimension(s)	One or two dimensions of an array, the maximum dimensions for the statement.

Elements in angle brackets are necessary elements of the statement. Braces enclose a choice of elements, one of which is required for the statement. Square brackets enclose optional elements of the statement.

A.4.1 Statements

CHAIN – Section 13.10

CHAIN <string> [[LINE] <line number>]

```
100 CHAIN "PROG3" LINE 75
200 CHAIN 'PROG3' LINE A
440 CHAIN 'PROG2'
550 CHAIN PROGRAM$ 2000
570 CHAIN 'PROG5' A%
```

CHANGE – Section 10.2

CHANGE { <array name> / <string variable> } TO { <string variable> / <array name> }

```
300 CHANGE A$ TO X
450 CHANGE STRING$ TO ARRAY%
700 CHANGE ARRAY% TO STRING$
```

CLOSE – Section 14.6

CLOSE [#] <expression> [,[#] <expression> ,...]

```
780 CLOSE #2%
450 CLOSE #1%,#2%,#5%
800 CLOSE #A% + B%
900 CLOSE #-CHAN% !Close without writing buffer to file
950 CLOSE #I% FOR I% = 1% TO 12% !Close all channels
```

COMMENT – Section 7.4

[<statement>] ! <message>

```
100 !This is a comment
150 PRINT !Perform a CR/LF
```

DATA – Section 9.2.3

DATA <value list>

```
1300 DATA 4.3, "STRING", 10, 1000, 1.45E9
```

DEF*, single line – Sections 9.8, 10.6, 11.6

DEF* FN<variable(arguments)> = <expression(arguments)>

```
120 DEF* FNA(X,Y,Z) = SQR(X^2% + Y^2% + Z^2%)
```

DEF*, multiple line – Section 13.1

DEF* FN <variable(arguments)>
<statements>
[LET] FN <variable> = <expression>
FNEND

```
300 DEF* FNF(M%) !Factorial function
310 IF M% = 0% OR M% = 1% &
              THEN FNF = 1% &
              ELSE FNF = M%*FNF(M% - 1%)
320 FNEND
```

NOTE

BASIC–PLUS supports DEF as well as DEF* in single and multi-line functions for compatibility with earlier versions. However, DEF* is preferred for compatibility with BASIC–PLUS–2.

DIM – Section 9.6

DIM <variable(dimension(s))> [,<variable(dimension(s))> ,...]

```
30 DIM A(20), B$(6,5), C%(99)
```

DIM (Virtual Array) – Section 16.1

DIM #<constant>,<stringvar(dimension(s))> [=<constant>] [,<var(dimension(s))> ,...]

```
70 DIM #4%, A$(100) = 32%, B(50,50), C%(30)
```

END – Section 9.10

END

```
32767 END
```

EXTEND – Section 7.5.1

EXTEND

```
10 EXTEND !Program in EXTEND mode
```

FIELD – Section 17.4.1

FIELD #<exp>,<exp1> AS <string var1>[,<exp2> AS <string var2>,...]

```
700 FIELD #2%, 10% AS A$, 20% AS B$, 5% AS C$
```

FOR – Section 9.5.1

FOR <variable> = <expression> TO <expression> [STEP <expression>]

```
200 FOR I% = 2% TO 40% STEP 2%
300 FOR T% = 0% TO T6% STEP I%
400 FOR N = A TO (C + S1)/A
```

FOR–WHILE, FOR–UNTIL – Section 13.5

FOR <variable> = <expression> [STEP <expression>] { WHILE / UNTIL } <condition>

```
450 FOR I = 1, STEP 3, WHILE I < X
470 FOR N% = 2% STEP 4% UNTIL N% > A% OR N% = B%
500 FOR B = 0, STEP B = 1, UNTIL B > B1
```

GET – Section 17.3

GET #<exp> [, { BLOCK <exp> / RECORD <exp> } ,COUNT <exp> ,USING <exp>]

```
140 GET #2%, BLOCK 99,
180 GET #4%, RECORD 50%
```

GOSUB – Section 9.9.1

GOSUB <line number>

```
190 GOSUB 2000
```

GOTO – Section 9.3

GOTO <line number>

```
100 GOTO 150
```

IF–THEN, IF–GOTO – Section 9.4

IF <condition> { THEN <statement> | THEN <line number> | GOTO <line number> }

```
50 IF A > B OR B > C THEN PRINT "No"
60 IF FNA(R) = B THEN 250
75 IF L < X^2% AND L <> 0, GOTO 1000
```

IF–THEN–ELSE – Section 13.4

IF <condition> { THEN <statement> | THEN <line number> | GOTO <line number> } [ELSE <statement> | ELSE <line number>]

```
200 IF B = A THEN PRINT 'EQUAL' ELSE PRINT 'NOT EQUAL'
300 IF A == N THEN 200 ELSE PRINT A \ STOP
400 IF FNA(R) = B &
          THEN GOTO 260 &
          ELSE LET B = B + FNA(R1) &
          \GOTO 1000
```

INPUT – Sections 9.2.2, 10.3.2, 15.2

INPUT [#<expression>,] <variable list>

```
100 INPUT A$
200 INPUT "Enter your name";A$
300 INPUT #1%, ITEM$, PART
```

INPUT LINE – Sections 10.3.3, 15.2.3

INPUT LINE [#<expression>,] <string variable>

```
300 INPUT LINE R$
400 INPUT LINE #8%, CUSTOMER$
```

KILL – Sections 5.4.2, 14.8

KILL <string>

```
100 KILL 'OLD.DAT'
200 KILL FILE.NAME$
```

LET – Section 9.1

[LET] <variable(s)> = <expression>

```
110 LET A% = 40%
120 B = 22.
130 C, F1, V(0) = 0 !Multiple assignment
```

LSET – Section 17.4.2

LSET <string var> [,<string var>,...] = <string>

```
100 LSET B$ = 'XYZ'
```

MAT Initialization – Section 12.5

MAT <matrix name> = $\left\{\begin{matrix}\text{ZER}\\ \text{CON}\\ \text{IDN}\end{matrix}\right\}$ [(dimension(s))]

```
100 DIM B(15,10), A(10), C%(5)
120 MAT C% = CON
130 MAT B = IDN(10,10)
140 MAT B = ZER(N,M)
```

MAT INPUT – Sections 12.4, 15.2.4

MAT INPUT [#<expression>,] <list of matrices>

```
150 DIM B$(40), F1%(35)
200 OPEN "DB2:MAT2.DAT" FOR INPUT AS FILE #3%
120 MAT INPUT #3%, B$, F1%
```

MAT PRINT – Sections 12.3, 15.1.3

MAT PRINT [#<expression>,] <matrix name>$\left[\begin{matrix},\\ ;\end{matrix}\right]$

```
150 DIM A(20), B%(15,30)
200 MAT PRINT A, !Print 20 elements, five on each line
250 MAT PRINT B%(10,25); !Print 10-by-25 subset of B%, packed
300 MAT PRINT #2%, A; !Print on channel 2
```

MAT READ – Section 12.2

MAT READ <list of matrices>

```
100 DIM A(20), B$(32), C%(15,10)
200 MAT READ A, B$(25), C%
```

NAME–AS – Sections 5.2.2, 14.7

NAME <string> AS <string>

```
455 NAME "NONAME" AS "FILE1.DAT/PR:40"
900 NAME 'DL1:MATRIX.DAT' AS 'MATA1.DAT<48>'
```

NEXT – Section 9.5.1

NEXT <variable>

```
100 NEXT I%
200 NEXT N
```

NOEXTEND – Section 7.5.1

NOEXTEND

```
50 NOEXTEND
50 NO EXTEND
```

ON ERROR GOTO – Section 13.7.1

ON ERROR GOTO [<line number>]

```
100 ON ERROR GOTO 9000
110 ON ERROR GOTO 0 !Disables error-handling routine
110 ON ERROR GOTO !Disables error-handling routine
```

ON–GOSUB – Section 13.3

ON <expression> GOSUB <list of line numbers>

```
230 ON FNC(M) GOSUB 2000, 2400, 3000
```

ON–GOTO – Section 13.2

ON <expression> GOTO <list of line numbers>

```
150 ON X% GOTO 170, 570, 430, 300
```

OPEN – Section 14.5

OPEN <string> [{FOR INPUT / FOR OUTPUT}] AS FILE [#] <expression>

[,RECORDSIZE <exp>] [,CLUSTERSIZE <exp>] [,FILESIZE <exp>] [,MODE <exp>]

```
100 OPEN 'LP:' FOR OUTPUT AS FILE #B1%
200 OPEN "FOO.DAT" AS FILE #3%
300 OPEN 'DM1:DATA.TR' FOR INPUT AS FILE #10%, RECORDSIZE 1024%
```

PRINT – Sections 9.2.1, 15.1

PRINT [[#<expression>,] <list>]

```
130 PRINT !Produces CR/LF
140 PRINT "Beginning of output:";I, A*I
150 PRINT #2%, 'OUTPUT TO DEVICE';N%
160 PRINT "Title: ";TITLE$, "Ref #";R$
```

PRINT–USING – Section 15.1.2

PRINT [#<expression>,] USING <string>, <list>

```
550 PRINT USING '##.##',AA
700 PRINT #7%, USING B$, A,B,C
```

PUT – Section 17.3

PUT #<exp> [, {BLOCK <exp> / RECORD <exp>} ,COUNT <exp> ,USING <exp>]

```
390 PUT #1%, COUNT 80%
```

RANDOMIZE – Section 9.7.4

RANDOM[IZE]

```
40 RANDOMIZE
100 RANDOM
```

READ – Section 9.2.3

READ <variable list>

```
100 READ A, B$, F1%, B(1%), R2
```

REM – Section 7.4

REM <message>

```
100 REM - This program plays solitaire
```

RESTORE – Section 9.2.4

RESTORE

```
100 RESTORE
```

RESUME – Section 13.7.2

RESUME [<line number>]

```
1000 RESUME 0 !Resume at line where error occurred
1000 RESUME !Equivalent to RESUME 0
650  RESUME 200 !Resume at line 200
```

RETURN – Section 9.9.2

RETURN

```
375 RETURN
```

RSET – Section 17.4.2

RSET <string var> [,<string var>,...] = <string>

```
250 RSET C$ = "12345"
```

SLEEP – Section 13.9

SLEEP <expression>

```
260 SLEEP 20% !Suspend job for 20 seconds
```

STOP – Sections 6.3.1, 9.11

STOP

```
180 STOP
```

UNLOCK – Section 17.7

UNLOCK #<expression>

```
500 UNLOCK #5%
```

UNTIL – Section 9.5.3

UNTIL <condition>

```
100 UNTIL LEN(A$) > 0%
```

WAIT – Section 13.9

WAIT <expression>

```
520 WAIT A%
```

WHILE – Section 9.5.2

WHILE <condition>

```
30 WHILE X < Y
```

A.4.2 Statement Modifiers

FOR – Section 13.6.3

<statement> FOR <variable> = <exp> TO <exp> [STEP <exp>]

[{ WHILE / UNTIL } <condition>]

```
175 LET B$(I%) = C$(I%) FOR I% = 1% TO J1%
190 READ A(I%) FOR I% = 0% TO 20% STEP J%
```

IF – Section 13.6.1

<statement> IF <condition>

```
100 PRINT T% IF T% > T1%
```

UNLESS – Section 13.6.2

<statement> UNLESS <condition>

```
350 PRINT A$ UNLESS Y% < 0%
```

UNTIL – Section 13.6.5

<statement> UNTIL <condition>

```
1080 IF B <> 0 THEN A(I%) = B UNTIL I% > K
```

WHILE – Section 13.6.4

<statement> WHILE <condition>

```
230 LET A(I%) = FNX(I%) WHILE A < 45.5
```

A.5 Reserved Keywords

BASIC–PLUS statement, function, variable, and option names are reserved keywords. This means that you cannot use them for your own variable or function names.

Table A–1 lists all language elements that are reserved for DIGITAL-supplied versions of BASIC–PLUS and BASIC–PLUS–2. Do not use any of the listed words as the name of a variable or function.

Table A–1: Reserved Keywords

ABS	DATA	FSP$	NOECHO
ABS%	DATE$	FSS$	NONE
ACCESS	DEF	GE	NOREWIND
ALLOW	DELETE	GET	NOSPAN
ALTERNATE	DENSITY*	GO	NOT
AND	DESC	GOSUB	NUL$
APPEND	DET	GOTO	NUM
AS	DIF$	GT	NUM$
ASCII	DIM	HT	NUM1$
ATN	DIMENSION	IDN	NUM2
BACK	DUPLICATES	IF	ON
BEL	ECHO	IMP	ONECHR
BLOCK	EDIT$	INDEXED	ONERROR
BLOCKSIZE	ELSE	INPUT	OPEN
BS	END	INSTR	OR
BUCKETSIZE	EQ	INT	ORGANIZATION
BUFFER	EQV	INV	OUTPUT
BUFSIZ	ERL	KEY	PEEK**
BY	ERN$	KILL	PI
CALL	ERR	LEFT	PLACE$
CCPOS	ERROR	LEFT$	POS
CHAIN	ERT$	LEN	PRIMARY
CHANGE	ESC	LET	PRINT
CHANGES	EXIT	LF	PROD$
CHR$	EXP	LINE	PUT
CLOSE	EXTEND**	LINPUT	QUO$
CLUSTERSIZE**	FF	LOC*	RAD$
COM	FIELD	LOG	RANDOM
COMMON	FILE	LOG10	RANDOMIZE
COMP%	FILE$	LSET	RCTRLC
CON	FILESIZE	MAGTAPE	RCTRLO
CONNECT	FILL	MAP	READ
CONTIGUOUS	FILL$	MAT	RECORD
COS	FILL%	MID	RECORDSIZE
COUNT	FIND	MID$	RECOUNT
CR	FIX	MODE	REF
CTRLC	FIXED	MODIFY	RELATIVE
CVT$$	FNEND	MOVE	REM
CVT$%	FNEXIT	NAME	RESET
CVT$F	FOR	NEXT	RESTORE
CVT%$	FORMAT$	NOCHANGES	RESUME
CVTF$	FROM	NODUPLICATES	RETURN

(continued on next page)

Table A–1: Reserved Keywords (Cont.)

RIGHT	SQR	TAPE	USING
RIGHT$	STATUS	TASK	VAL
RND	STEP	TEMPORARY	VAL%
RSET	STOP	THEN	VALUE
SCRATCH	STR$	TIME	VARIABLE
SEG%	STREAM	TIME$	VIRTUAL
SEQUENTIAL	STRING$	TO	VT
SGN	SUB	TRM$	WAIT
SI	SUBEND	TRN	WHILE
SIN	SUBEXIT	UNDEFINED	WINDOWSIZE
SLEEP	SUM$	UNLESS	WRITE
SO	SWAP%	UNLOCK	WRKMAP
SP	SYS**	UNTIL	XLATE
SPACE$	TAB	UPDATE	XOR
SPAN	TAN	USEROPEN	ZER
SPEC			

*Reserved word, included for compatibility with DECSYSTEM 20
**Supported on RSTS/E only

Appendix B
Command Summary

Table B–1 briefly describes the BASIC–PLUS keyboard monitor commands. For details on the syntax and use of these commands, see Part I of this manual and the *RSTS/E System User's Guide.* (Commands marked with an asterisk (*) are described in the *RSTS/E System User's Guide.*) Table B–2 summarizes control characters and terminal keys.

Table B–1: BASIC–PLUS Commands

Command	Description
APPEND	Merges the contents of a previously saved source program with the current program. The BASIC–PLUS APPEND command differs from the DCL APPEND command. The DCL APPEND command appends one file to the end of another. See the *RSTS/E DCL User's Guide* for more information.
ASSIGN*	Reserves an I/O device for use by a job. Only the job that issues the ASSIGN command can use the device. ASSIGN also assigns a logical name to a device, assigns an account to the @ character, and sets a default protection code. The BASIC–PLUS ASSIGN command differs from the DCL ASSIGN command. The DCL ASSIGN command only assigns a logical name to a device. See the *RSTS/E DCL User's Guide* for more information.
BYE*	Logs you out; closes and saves any open files. After you type BYE and press the RETURN key, the system displays: Confirm: You have five options: ? Displays help on valid responses to the "Confirm" prompt. Y Normal logout. N No logout; cancels the BYE command. I Delete files individually before logging out. F Fast logout. To request one of these options in the BYE command, type BYE, a slash, and one of the valid responses (BYE/F, for example).

(continued on next page)

Table B–1: BASIC–PLUS Commands (Cont.)

Command	Description
CAT CATALOG	Displays your file directory. The default device is the system disk, but you can specify another device after the word CAT or CATALOG.
CCONT	For privileged users. Same as CONT command, but detaches the job from the terminal.
COMPILE	Saves a translated image of the current program in a disk file. The default file name is the current program name; the default file type is .BAC.
CONT	Continues execution of the current program after execution of a STOP statement.
DEASSIGN*	Releases a device for use by other jobs. DEASSIGN also releases a logical name for a device, cancels the association between an account and the @ character, and changes the current default protection code back to the system default. The system performs an automatic DEASSIGN when you enter the BYE command. The BASIC–PLUS DEASSIGN command differs from the DCL DEASSIGN command. The DCL DEASSIGN command only releases a logical name for a device. See the *RSTS/E DCL User's Guide* for more information.
DELETE	Removes one or more lines from the program in memory. After the word DELETE, type the line number of the line to be deleted or two line numbers separated by a dash (–). You can specify several single lines or ranges of lines. Use commas to separate line numbers or line number ranges. Typing DELETE with no line numbers deletes all lines from your current program. The BASIC–PLUS DELETE command differs from the DCL DELETE command. The DCL DELETE command deletes a file from a directory. See the *RSTS/E DCL User's Guide* for more information.
EXIT	Clears memory and returns control to the job keyboard monitor.
EXTEND	Puts BASIC–PLUS in EXTEND mode. You can write and run programs that include EXTEND mode features.
HELLO*	Tells RSTS/E that you want to log in. The system prompts you for a project-programmer number and password. You can also use HELLO to attach a detached job to the terminal or change accounts without logging out.
KEY	Reenables terminal echo after a TAPE command. (Use this command only with ASR33 terminals that have a low-speed paper-tape reader.) Press the LINE FEED key before entering the KEY command in case the last line input did not end with a carriage return/line feed. Use the LINE FEED or ESCAPE key (not RETURN) to enter the KEY command. The system does not treat a carriage return character as a delimiter when the terminal is in tape mode.

(continued on next page)

Table B–1: BASIC–PLUS Commands (Cont.)

Command	Description
LENGTH	Returns the length of your current program in 1K increments, along with its maximum allowed size. For example, if the current program is between 6K and 7K and the maximum size is 16K, BASIC–PLUS displays: `7(16)K of memory used`
LIST	Displays all or part of the program currently in memory. The word LIST by itself displays your entire program. LIST followed by one line number displays that line; LIST followed by two line numbers separated by a dash (–) displays the lines between and including the indicated lines. You can display several single lines or ranges of lines. Use commas to separate line numbers or line number ranges.
LISTNH	Same as LIST, but does not print the header that contains the program name and current date and time.
NEW	Clears your memory area, names a new program, and lets you enter a new program at the terminal. The default program name is NONAME.
NOEXTEND	Puts BASIC–PLUS in NOEXTEND mode. EXTEND mode features are no longer available unless the program contains an EXTEND statement.
OLD	Retrieves a saved source program from disk and places it in memory. By default, OLD retrieves NONAME.BAS in your account on the public structure.
REASSIGN*	Transfers control of a device to another job.
RENAME	Changes the name of the program currently in memory. The BASIC–PLUS RENAME command differs from the DCL RENAME command. The DCL RENAME command renames files. See the *RSTS/E DCL User's Guide* for more information.
REPLACE	Copies the source program currently in memory into a disk file. The default file name is the current program name; the default file type is .BAS. Unlike the SAVE command, REPLACE replaces an existing file with the same name.
RUN	Executes the program in memory. If you type a file specification after the word RUN, the system loads the file from disk, translates it if necessary, and executes it.
RUNNH	Executes the program in memory without printing the header that contains the program name and current date and time. You can use RUNNH to run the current program only.
SAVE	Copies the source program currently in memory into a disk file. The default file name is the current program name; the default file type is .BAS. SAVE does not replace an existing file with the same name.
SCALE	Sets the scale factor to a specified value. When you do not specify a value, SCALE displays the current and pending scale factors.

(continued on next page)

Table B–1: BASIC–PLUS Commands (Cont.)

Command	Description
TAPE	Disables terminal echo while the terminal's low-speed reader reads a paper tape into the system. (Use this command only with ASR33 terminals that have a low-speed paper-tape reader.) Use the NEW command before the TAPE command. NEW causes BASIC–PLUS to expect input of a source program. After entering the TAPE command, insert the tape in the low-speed reader and set the reader's control switch to START. After you type the TAPE command, the system ignores RUBOUT characters and does not treat carriage return characters as delimiters. The program is not printed on the terminal as the system reads it, but error messages are printed. Use the KEY command to reenable terminal echo when you finish using the low-speed reader.
UNSAVE	Deletes a file from a directory. The default file name is your current program name; the default file type is .BAS.

Table B–2: Control Characters and Terminal Keys

Key	Function
CTRL/C	Halts execution of the current program and returns control to the job keyboard monitor. Echoes on the terminal as "^C".
CTRL/O	Stops and restarts terminal output while a program is running.
CTRL/Q	Resumes terminal output suspended by CTRL/S while a program is running. You can use CTRL/Q only if the terminal STALL characteristic is set.
CTRL/R	Redisplays the current terminal line.
CTRL/S	Suspends terminal output while a program is running. You can use CTRL/S only if the terminal STALL characteristic is set.
CTRL/U	Deletes the current terminal line. CTRL/U does not erase characters from the screen. Instead, it echoes "^U" and moves the cursor to the next line.
CTRL/Z	Is an end-of-file marker.
DELETE	Erases the last character typed.
ESCAPE ALTMODE	Sends a typed line to the system for processing. Echoes on your terminal as a "$" and does not perform a carriage return/line feed.
FORM FEED CTRL/L	Sends a typed line to the system for processing. Performs a form feed operation on the terminal.
LINE FEED	Continues the current program line on another terminal line. Performs a line feed/carriage return operation. It is recommended that you use the ampersand (&)/RETURN key combination instead of the LINE FEED key to continue the current program line on another terminal line. You can use the ampersand/RETURN key combination only in EXTEND mode.
NO SCROLL	Performs the same function as CTRL/S and CTRL/Q on a VT100 terminal. The terminal STALL characteristic must be set.
RETURN	Sends a typed line to the system for processing. Performs a carriage return/line feed operation on the terminal.
RUBOUT	Erases the last character typed and echoes erased characters inside backslashes.
TAB	Moves the cursor to the next tab stop on the terminal line. By default, tab stops are eight spaces apart.

Appendix C
Error Messages

C.1 Interpretation of Error Messages

Messages in RSTS/E are generated both for BASIC–PLUS errors and RSTS/E errors. These messages are called *RSTS/E error messages* and are described as one set. The BASIC–PLUS errors cover conditions that occur during translation and at run time, such as a violation of the syntax rules (?Syntax error) and referencing an element of an array beyond the defined limits (?Subscript out of range). The RSTS/E errors involve operating system conditions, such as failing to locate the file or account specified (?Can't find file or account) and requesting the hardware to perform a function for which it is not ready (?Device hung or write locked). Tables C–4 and C–5 describe the RSTS/E error messages.

In most cases, if you are not trapping errors (that is, an ON ERROR GOTO statement is not in effect), BASIC–PLUS stops running the program. It prints the error message and the line number of the BASIC–PLUS statement that was being executed when the error occurred. The following sample printout shows the procedure:

```
10      OPEN 'Z' FOR INPUT AS FILE 1%
RUNNH
?Can't find file or account at line 10

Ready
```

As the Ready prompt indicates, control returns to the system.

One exception to this procedure occurs when you execute an INPUT statement at the job's console terminal and error trapping is not in effect. The system generates the error message and executes the statement again, as shown in the following sample printout:

```
10      ON ERROR GOTO 0 \ INPUT 'INTEGER VALUE';A%
RUNNH
INTEGER VALUE? C
%Data format error at line 10
INTEGER VALUE?
```

With error trapping disabled at line 10, an invalid response to the INPUT statement causes the system to print the error message, clear the error condition, and execute the statement again.

Associated with each message is an error variable called ERR. Whenever an error occurs with trapping in effect, the system checks the error variable (which is a decimal number in the range 0 to 127). An error with a number between 1 and 70 causes the system to transfer control to the line number indicated in the ON ERROR GOTO statement. The system does not print the error message. Your program is able to check the ERR variable and perform a recovery procedure.

If the error number is between 71 and 127, the system does not transfer control to the recovery routine but prints the message and returns control to the system. (Error number 0 is reserved to identify the system installation name.)

Because a BASIC–PLUS program can recover from certain errors, this appendix lists errors in two categories—*recoverable* and *nonrecoverable*. The recoverable error messages are in ascending order of their error numbers. A program can use these error numbers to differentiate errors. Nonrecoverable errors are in alphabetical order without error numbers, because a program cannot use these numbers in an error handling routine.

The first character position of each message indicates the severity of the error. Table C–1 describes this standard.

Table C–1: Severity Standard in Error Messages

Character	Severity	Meaning
%	Warning	Execution of the program can continue but may not generate the expected results.
?	Fatal	Execution cannot continue unless you remove the cause of the error. No space or tab is allowed after the question mark.
	Information	A message beginning with neither a question mark nor a percent sign is for information only.

In the error message descriptions in Tables C–4 and C–5, the abbreviations shown in Table C–2 denote special characteristics of the error.

Table C–2: Special Abbreviations for Error Descriptions

Abbreviation	Meaning
C	Continue. If an ON ERROR GOTO statement is not in effect, execution continues but with the conditions described.
SPR	Software Performance Report. This error should occur only under the conditions described. If it occurs under any other conditions, you should document the conditions under which the error occurred and have the appropriate person at your site send an SPR to DIGITAL.

If an error occurs in a program with no error trapping in effect, shown as "C" in Table C–2, BASIC–PLUS prints the error message and line number but continues running the program. The following sample printout shows the procedure:

```
100      ON ERROR GOTO 0 \ A% = 32768.
200      PRINT A%
RUNNH
%Integer error at line 100
 0

Ready
```

The attempt to compute a value outside the range for integers generated an "%Integer error" at line 100. After BASIC–PLUS prints the error message, processing continues but with the conditions described in the error meaning. BASIC–PLUS substitutes 0 for the erroneously computed value.

The number of RSTS/E error messages is restricted to 127. Because of this restriction, certain error messages have multiple meanings. The specific meaning of an error message depends on the operation you are performing when the error condition occurs. For example, if the system attempts a file access and the designated file cannot be located, RSTS/E generates the "?Can't find file or account" error (ERR=5). That same error condition, however, applies to other, generically similar access operations.

Thus, if a program attempts to send a message to another program and the system cannot find the proper entry in the system table of eligible receivers, RSTS/E returns error number 5. Though the second failure does not involve a file access error, it too is classified as an access failure.

Certain RSTS/E errors, although classified as user-recoverable, cannot be trapped by a program. Table C–3 lists these errors.

Table C–3: Non-Trappable Errors in Recoverable Class

ERR	Message Printed
34	?Reserved instruction trap
36	?SP stack overflow
37	?Disk error during swap
38	?Memory parity failure

These errors involve special conditions that your program cannot control and that should not occur on a normal system. For example, the "?Disk error during swap" error indicates a hardware problem. The system does not return control to the program. The error condition itself, however, can be either transient or recurring.

Bring these errors to the attention of your system manager for further investigation. These errors are recoverable in the strict sense that the monitor can take corrective action. However, the BASIC–PLUS run-time system does not return control to your program.

Table C–4: User-Recoverable Error Messages

ERR	Message Printed	Meaning
0	(system installation name)	The error code 0 is associated with the system installation name. System programs use this to print identification lines.
1	?Bad directory for device	1. The directory of the device referenced is in an unreadable format. 2. The magnetic tape label format on tape differs from the system-wide default format, the current job default format, or the format specified in the OPEN statement. Use the ASSIGN command to set the correct format default or change the format specification in the MODE option of the OPEN statement.
2	?Illegal file name	1. The file name or type specified is not acceptable. It contains unacceptable characters or violates the file specification format. 2. The CCL command to be added begins with a number or contains a character other than A through Z, 0 through 9, or at sign (@).
3	?Account or device in use	1. The account to be deleted has one or more files and must be zeroed before being deleted. 2. Reassigning or dismounting of the device cannot be done because the device is open or has one or more open files. 3. The run-time system to be deleted is in use. 4. Output to a pseudo keyboard cannot be done unless the device is in KB wait state. 5. An echo control field cannot be declared while another field is active. 6. The CCL command to be added already exists.
4	?No room for user on device	You have already used the allowed storage space; the device as a whole is too full to accept further data.
5	?Can't find file or account	Either the file or account number specified was not found on the device specified, or the CCL command to be deleted does not exist.

(continued on next page)

Table C–4: User-Recoverable Error Messages (Cont.)

ERR	Message Printed	Meaning
6	?Not a valid device	The device specification supplied is not valid for one of the following reasons: 1. The unit number or its type is not configured on the system. 2. The specification is logical and untranslatable because a physical device is not associated with it.
7	?I/O channel already open	You tried to open one of the twelve I/O channels that the program had already opened. (SPR)
8	?Device not available	The specified device exists on the system, but you cannot assign or open it for one of the following reasons: 1. The device is currently reserved by another job. 2. The device requires privileges for ownership that you do not have. 3. The system manager has disabled the device or its controller. 4. The device is a keyboard line for pseudo keyboard use only.
9	?I/O channel not open	You tried to perform I/O on one of the twelve channels that the program has not previously opened.
10	?Protection violation	You cannot perform the requested operation because the operation is illegal (such as input from a line printer) or because you do not have the privileges necessary (such as deleting a protected file).
11	?End of file on device	You tried to perform input beyond the end of a data file, or a BASIC–PLUS source file is called into memory that does not contain an END statement.
12	?Fatal system I/O failure	An I/O error has occurred on the system level. You have no guarantee that the last operation has been performed. This error is caused by a hardware condition. Report such occurrences to the system manager.
13	?Data error on device	One or more characters may have been transmitted incorrectly due to a parity error, bad punch combination on a card, or similar error.

(continued on next page)

Table C–4: User-Recoverable Error Messages (Cont.)

ERR	Message Printed	Meaning
14	?Device hung or write locked	Check the hardware condition of the device you are requesting. Possible causes of this error include a line printer out of paper or high-speed reader being off-line.
15	?Keyboard WAIT exhausted	Time that the WAIT statement requests has been exhausted with no input received from the specified keyboard.
16	?Name or account now exists	Either you tried to rename a file with the name of a file that already exists, or the system manager tried to insert an account number that is already in the system.
17	?Too many open files on unit	Only one open DECtape output file is permitted per DECtape drive. Only one open file per magnetic tape drive is permitted.
18	?Illegal SYS( ) usage	Illegal use of the SYS system function.
19	?Disk block is interlocked	The requested disk block segment is already in use (locked) by some other user.
20	?Pack IDs don't match	The identification code for the specified disk pack does not match the identification code already on the pack.
21	?Disk pack is not mounted	No disk pack is mounted on the specified disk drive.
22	?Disk pack is locked out	The disk pack specified is mounted but is temporarily disabled.
23	?Illegal cluster size	The specified cluster size is unacceptable. The cluster size must be a power of 2. For a *file* cluster, the size must be equal to or greater than the pack cluster size and must not be greater than 256. For a *pack* cluster, the size must be equal to or greater than the device cluster size and must not be greater than 16. The device cluster size is fixed by type.
24	?Disk pack is private	You do not have access to the specified private disk pack.
25	?Disk pack needs 'CLEANing'	Nonfatal disk mounting error; run the ONLCLN system program.
26	?Fatal disk pack mount error	Fatal disk mounting error. Disk cannot be successfully mounted.
27	?I/O to detached keyboard	I/O was attempted to a hung-up dataset or to the previous, but now detached, console keyboard for the job.

(continued on next page)

Table C–4: User-Recoverable Error Messages (Cont.)

ERR	Message Printed	Meaning
28	?Programmable ^C trap	A CTRL/C was typed while an ON ERROR GOTO statement was in effect and programmable CTRL/C trapping was enabled.
29	?Corrupted file structure	Fatal error in CLEAN operation.
30	?Device not file structured	An attempt is made to access a device other than a disk, DECtape, or magnetic tape device as a file-structured device. This error occurs, for example, when you attempt to get a directory listing of a non-directory device.
31	?Illegal byte count for I/O	This error has two possible causes: 1. The buffer size you specified in the RECORDSIZE option of the OPEN statement or the COUNT option of the PUT statement is not a multiple of the block size of the device you are using for I/O, or is illegal for the device. 2. You tried to run a compiled file that has improper size due to incorrect transfer procedure.
32	?No buffer space available	You accessed a file and the monitor required one small buffer to complete the request, but there is no buffer available. If the program is sending messages, two conditions are possible. The first occurs when a program sends a message and the receiving program has exceeded the pending message limit. The second occurs when a sending program attempts to send a message and a small buffer is not available for the operation.
33	?Odd address trap	This error occurs when you attempt to address nonexistent memory or an odd address with the PEEK function. If you get this error for any other reason, report it to your system manager.
34	?Reserved instruction trap	An attempt is made to execute an illegal or reserved instruction or an FPP instruction when floating-point hardware is not available. (See discussion at beginning of appendix.)
35	?Memory management violation	You specified an illegal monitor address in the PEEK function. If you get this error for any other reason, report it to your system manager.

(continued on next page)

Table C–4: User-Recoverable Error Messages (Cont.)

ERR	Message Printed	Meaning
36	?SP stack overflow	An attempt was made to extend the hardware stack beyond its legal size. (See discussion at beginning of appendix.) (SPR)
37	?Disk error during swap	A hardware error occurs when your job is swapped into or out of memory. The contents of your job area are lost, but the job remains logged into the system and is reinitialized to run the NONAME program. Report such occurrences to the system manager. (See discussion at beginning of appendix.)
38	?Memory parity failure	A parity error was detected in the memory occupied by this job. (See discussion at beginning of appendix.)
39	?Magtape select error	When access to a magnetic tape drive was attempted, the selected unit was found to be off line.
40	?Magtape record length error	When performing input from magnetic tape, the record on magnetic tape was longer than the buffer designated to handle the record.
41	?Non-res run-time system	The run-time system referenced has not been loaded into memory and is therefore nonresident.
42	?Virtual buffer too large	Virtual array buffers must be 512 bytes long.
43	?Virtual array not on disk	A nondisk device is open on the channel on which the virtual array is referenced.
44	?Matrix or array too big	Memory array size is too large.
45	?Virtual array not yet open	You tried to use a virtual array before opening the corresponding disk file.
46	?Illegal I/O channel	You tried to open a file on an I/O channel outside the range of the integer numbers 1 to 12.
47	?Line too long	The buffer overflows because of an attempt to input a line longer than 255 characters. (This includes any line terminator.)
48	%Floating point error	You tried to use a computed floating-point number outside the range 1E–38 <n<1E38 excluding zero. If no transfer to an error handling routine is made, zero is returned as the floating-point value. (C)

(continued on next page)

Table C–4: User-Recoverable Error Messages (Cont.)

ERR	Message Printed	Meaning
49	%Argument too large in EXP	Acceptable arguments are within the approximate range –89<arg<+88. The value returned is zero. (C)
50	%Data format error	A READ or INPUT statement detected data in an illegal format. For example, 1..2 is an improperly formed number; 1.3 is an improperly formed integer; "HELLO" "THERE" is an illegal string. (C)
51	%Integer error	You tried to use a computed integer outside the range –32768<n<32767. For example, you tried to assign to an integer variable a floating-point number outside the integer range. If no transfer to an error handling routine is made, zero is returned as the integer value. (C)
52	?Illegal number	Integer overflow or underflow, or floating-point overflow. The range for integers is –32768 to +32767; for floating-point numbers, the upper limit is 1E38. (For floating-point underflow, the "%Floating point error" (ERR=48) is generated.)
53	%Illegal argument in LOG	Negative or zero argument to LOG function. Value returned is the argument as passed to the function. (C)
54	%Imaginary square roots	You tried to take the square root of a number less than zero. The value returned is the square root of the absolute value of the argument. (C)
55	?Subscript out of range	You tried to reference an array element beyond the number of elements created for the array when it was dimensioned.
56	?Can't invert matrix	You tried to invert a singular or nearly singular matrix.
57	?Out of data	The DATA list was exhausted and a READ requested additional data.
58	?ON statement out of range	The index value in an ON-GOTO or ON-GOSUB statement is less than one or is greater than the number of line numbers in the list.
59	?Not enough data in record	An INPUT statement did not find enough data in one line to satisfy all the specified variables.
60	?Integer overflow, FOR loop	The integer index in a FOR loop attempted to go beyond 32766 or below –32767.

(continued on next page)

Table C–4: User-Recoverable Error Messages (Cont.)

ERR	Message Printed	Meaning
61	%Division by 0	Attempt in your program to divide some quantity by zero. If no transfer is made to an error handling routine, the result is 0. (C)
62	?No run-time system	The run-time system referenced has not been added to the system list of run-time systems.
63	?FIELD overflows buffer	You tried to use FIELD to allocate more space than exists in the specified buffer.
64	?Not a random access device	You tried to perform random access I/O to a nonrandom access device.
65	?Illegal MAGTAPE( ) usage	Improper use of the MAGTAPE function.
66	?Missing special feature	Your program uses a BASIC–PLUS feature not present on the given installation.
67	?Illegal switch usage	This error has two possible causes: 1. A CCL command contains an error in an otherwise valid CCL switch. (For example, use of the /SI:n switch without a value for n or a colon; or specification of more than one of the same type of CCL switch.) 2. A file specification switch is not the last element in a file specification or is missing a colon or an argument.

Table C–5: Nonrecoverable Error Messages

Message Printed	Meaning
?Arguments don't match	Arguments in a function call do not match, in number or in type, the arguments defined for the function.
?Bad line number pair	Line numbers specified in a LIST or DELETE command were formatted incorrectly.
?Bad number in PRINT-USING	Format specified in the PRINT-USING string cannot be used to print one or more values.
?Can't CONTinue	Program was stopped or ended at a spot from which execution cannot be resumed with CONT or CCONT.
?Data type error	Incorrect usage of floating-point, integer, or string variable or constant where some other data type was necessary.

(continued on next page)

Table C–5: Nonrecoverable Error Messages (Cont.)

Message Printed	Meaning
?DEF without FNEND	A second DEF statement was encountered in the processing of a user function without an FNEND.
?End of statement not seen	Statement contains too many elements to be processed correctly.
?Error text lookup failure	An I/O error occurred while the system was attempting to retrieve an error message. Possible cause could be that the device containing the system error file (ERR.SYS) is off-line or the system error file contains a bad block.
?Execute only file	Attempt was made to add, delete, or list a statement in a translated (.BAC) file.
?Expression too complicated	This error usually occurs when parentheses have been nested too deeply. The depth allowed depends on the individual expression.
?File exists-RENAME/REPLACE	A file of the name specified in a SAVE command already exists. To save the program with the name specified, use REPLACE, or use RENAME followed by SAVE.
?FNEND without DEF	A FNEND statement was encountered in your program before a DEF statement was seen.
?FNEND without function call	An FNEND statement was encountered in your program before a function call was executed.
?FOR without NEXT	A FOR statement was encountered in your program without a corresponding NEXT statement to terminate the loop.
?Illegal conditional clause	Incorrectly formatted conditional expression.
?Illegal DEF nesting	The range of one function definition crosses the range of another function definition.
?Illegal dummy variable	One of the variables in the dummy variable list of user-defined functions is not a legal variable name.
?Illegal expression	Double operators, missing operators, mismatched parentheses, or some similar error was found in an expression.
?Illegal FIELD variable	The FIELD variable specified is unacceptable.
?Illegal FN redefinition	Attempt was made to redefine a user function.
?Illegal function name	Attempt was made to define a function with a function name of incorrect format.
?Illegal IF statement	Incorrectly formatted IF statement.
?Illegal in immediate mode	You entered a statement in immediate mode that can only be executed as part of a program.
?Illegal line number(s)	Line number reference outside the range $1<n<32767$.

Table C–5: Nonrecoverable Error Messages (Cont.)

Message Printed	Meaning
?Illegal mode mixing	String and numeric operations cannot be mixed.
?Illegal statement	Attempt was made to execute a statement that did not translate without errors.
?Illegal symbol	An unrecognizable character was encountered. For example, a line consisting of a # character.
?Illegal verb	The verb portion of the statement cannot be recognized.
%Inconsistent function usage	A function is defined with a certain number of arguments but is referenced elsewhere with a different number of arguments. Fix the reference to match the definition and reload the program to reset the function definition.
%Inconsistent subscript use	A subscripted variable is being used with a different number of dimensions from the number with which it was originally defined.
?Literal string needed	A variable name was used where a numeric or character string was necessary.
?Matrix dimension error	Attempt was made to dimension a matrix to more than two dimensions, or an error was made in the syntax of a DIM statement.
?Matrix or array without DIM	A matrix or array element was referenced beyond the range of an implicitly dimensioned matrix.
?Maximum memory exceeded	This error has two possible causes: 1. During an OLD operation, the job's private memory size maximum was reached. 2. While running a program, the system required more memory for string or I/O buffer space, and the job's private memory size maximum or the system maximum (16K words for BASIC–PLUS) was reached.
?Modifier error	This error has two possible causes: 1. Attempt was made to use one of the statement modifiers (FOR, WHILE, UNTIL, IF or UNLESS) incorrectly. 2. An OPEN statement modifier, such as a RECORDSIZE, CLUSTERSIZE, FILESIZE, or MODE option, is not in the correct order.
?NEXT without FOR	A NEXT statement was encountered in your program without a previous FOR statement.
?No logins	Message printed if the system is full and cannot accept additional users, or if further logins are disabled by the system manager.

(continued on next page)

Table C–5: Nonrecoverable Error Messages (Cont.)

Message Printed	Meaning
?Not enough available memory	An attempt was made to load a nonprivileged compiled program that is too large to run given the job's private memory size maximum. The program must be made privileged in order to expand above a private memory size maximum, or the system manager must increase the job's private memory size maximum to accommodate the program.
?Number is needed	A character string or variable name was used where a number was necessary.
?1 or 2 dimensions only	Attempt was made to dimension a matrix to more than two dimensions.
?ON statement needs GOTO	A statement beginning with ON does not contain a GOTO or GOSUB clause.
Please say HELLO	Message printed by the LOGIN system program. A user who was not logged into the system has typed something other than a legal, logged-out command.
?Please use the RUN command	A transfer of control (as in a GOTO, GOSUB or IF-GOTO statement) cannot be performed from immediate mode.
?PRINT–USING buffer overflow	Format specified contains a field too large to be manipulated by the PRINT–USING statement.
?PRINT–USING format error	An error was made in the construction of the string used to supply the output format in a PRINT–USING statement.
?Program lost–Sorry	A fatal system error caused your program to be lost. This error can indicate hardware problems or use of an improperly compiled program. See Section C.2 for more information.
?Redimensioned array	Usage of an array or matrix within your program has caused BASIC–PLUS to redimension the array implicitly.
?RESUME and no error	A RESUME statement was encountered where no error had occurred to cause a transfer into an error handling routine with the ON ERROR GOTO statement.
?RETURN without GOSUB	RETURN statement is encountered in your program when a previous GOSUB statement was not executed.
%SCALE factor interlock	1. You set a new scale factor and then executed a program that was translated using a different scale factor. The program runs, but BASIC–PLUS uses the scale factor in effect when the program was translated. To cause BASIC–PLUS to translate the program with the new scale factor, use REPLACE and OLD.

(continued on next page)

Table C–5: Nonrecoverable Error Messages (Cont.)

Message Printed	Meaning
%SCALE factor interlock (Cont.)	2. You set a new scale factor and then entered an immediate mode statement. Immediate mode statements are always translated using the current scale factor. The new scale factor will take effect when you use the NEW or OLD command or run a program from its source file. See Section 11.10.2 for more information. (C)
?Statement not found	Reference is made in the program to a line number that is not in the program.
Stop	STOP statement was executed. You can usually continue program execution by typing CONT and the RETURN key.
?String is needed	A number or variable name was used where a character string was necessary.
?Syntax error	BASIC–PLUS statement was incorrectly formatted.
?Too few arguments	The function has been called with a number of arguments not equal to the number defined for the function.
?Too many arguments	A user-defined function can have up to five arguments only.
?Undefined function called	BASIC–PLUS interpreted some statement component as a function call for which there is no defined function (system or user).
?What?	You entered a command or immediate mode statement that BASIC–PLUS cannot process. Illegal verb or improper format error most likely.
?Wrong math package	Program was compiled on a system with either the two-word or four-word math package, and an attempt is made to run the program on a system with the opposite math package. Recompile the program using the math package of the system on which it will be run.

C.2 The "?Program Lost–Sorry" Error

The "?Program lost–Sorry" error occurs when BASIC–PLUS tries to run a program and cannot. BASIC–PLUS clears the job image from memory and returns control to the user. If possible, BASIC–PLUS prints a second message that provides more information about what caused the program to be lost. In several cases, however, only the "?Program lost–Sorry" message is printed, and the system manager must check the error log to determine the cause. Always report a "?Program lost–Sorry" message and its associated message (if printed) to your system manager.

The "?Program lost–Sorry" error has four possible causes:

1. A checksum error occurs on a .BAC file. (A checksum error is usually the result of a hardware problem.)
2. An unrecoverable disk error occurs while BASIC–PLUS is reading a .BAC file.
3. BASIC–PLUS tries to load a .BAC file of incorrect size.
4. BASIC–PLUS tries to run a file whose stored version number does not match the current BASIC–PLUS run-time system's version number.

You can often recover by recompiling the program from its source file and running it again. To recompile the program:

1. Use the OLD command to translate the program from its source file. OLD places the translated program in memory.
2. Use the COMPILE command to create a new .BAC file that contains the translated image.

The next four sections describe each of the possible causes in more detail.

C.2.1 Checksum Error on a .BAC File

A "checksum" is a numeric quantity that is used to detect errors. When you save a translated program in a disk file with the COMPILE command, BASIC–PLUS computes a checksum and stores it in the file. BASIC–PLUS computes another checksum when it loads the .BAC file from disk. An error occurs if the computed and stored checksums do not match.

If the checksums are not equal, BASIC–PLUS produces an error to be logged by the RSTS/E monitor, returns the "?Program lost–Sorry" error to the user, and aborts program execution.

Checksum errors are usually caused by a disk error. The disk error may have occurred when you created the .BAC file or it may have occurred while BASIC–PLUS was reading the .BAC file into memory.

You can usually recover by recompiling the program and running it again.

C.2.2 Unrecoverable Disk Error Reading a .BAC File

The "?Program lost–Sorry" error also results when an unrecoverable disk error occurs while BASIC–PLUS is loading a .BAC file into memory. Unrecoverable disk errors can result from bad disk blocks, dust, problems with the disk drive, or a transient hardware problem in the disk subsystem. Sometimes these errors produce a warning message such as "?Disk error during swap", which is logged in the system error logging file.

Recompiling the program may correct the problem. Be sure to report the problem to your system manager.

C.2.3 Incorrect .BAC File Size

A .BAC file must be between 2K and 16K words (inclusive). In addition, the number of blocks in the file must be an integer that is one less than a multiple of 4.

If the size of the .BAC file does not follow these rules, BASIC–PLUS prints two messages when it tries to load the file into memory: "?Program lost–Sorry" and "?Illegal byte count for I/O". These errors are not logged in the system error logging file.

To correct the problem, recompile the program.

C.2.4 Unmatched Version Numbers

When you use COMPILE to save a translated program, BASIC–PLUS writes the current version number of the BASIC–PLUS run-time system into the .BAC file. When BASIC–PLUS runs or chains to a .BAC file, it checks the version number stored in the file against the version number of the run-time system being used. If the version numbers do not match, the "?Program lost–Sorry" error results.

Consult the RSTS/E Release Notes to find out whether recompilation of current programs is necessary for a new version of BASIC–PLUS.

C.3 Reporting Software Problems

Report problems with DIGITAL software to your system manager. Your system manager will determine whether the problem needs to be reported to DIGITAL.

Appendix D
BASIC–PLUS Character Set

BASIC–PLUS programs are composed of:

- The letters A through Z in both upper- and lowercase
- The digits 0 through 9
- Spaces
- Tabs
- The special symbols and keys listed in Table D–1

Table D–1: Special Symbols and Keys

Symbol or Key	Use
$	Suffix for string variables and functions; see Section 8.3 and Chapter 10.
%	Suffix for integer variables and functions; see Section 8.3 and Chapter 11.
.	Indicates the decimal point in floating-point numbers and numeric strings; see Section 8.3. The period is a valid character in EXTEND mode variable names; see Section 8.3.2.1. The period also separates the file name and type in a file specification; see the *RSTS/E System User's Guide*.
'	Delimiter for string constants (text strings); see Section 8.3.1.3.
"	Delimiter for string constants (text strings); see Section 8.3.1.3.
!	Begins a comment; see Section 7.4.
\	Separates multiple statements on one program line; see Section 7.3.1.
:	Separates multiple statements on one program line; accepted for compatibility with previous versions of BASIC–PLUS. The backslash (\) is the preferred character.

(continued on next page)

Table D–1: Special Symbols and Keys (Cont.)

Symbol or Key	Use
#	Indicates an I/O channel number; see Section 14.5. The # is also a formatting character in the PRINT–USING statement; see Section 15.1.2.
,	Formatting character in PRINT statement; see Section 9.2.1.2. The comma is also part of the syntax for several BASIC–PLUS statements. For example, you use commas between variable names in the INPUT statement and between options in statements such as OPEN, GET, and PUT.
;	Formatting character in PRINT statement; see Section 9.2.1.2.
&	Part of the &/RETURN key combination, which indicates that a statement is continued on the next terminal line; legal only in EXTEND mode. See Section 7.3.2.
LF	Indicates that a statement is continued on the next terminal line; see Section 7.3.2. (The &/RETURN key combination is the preferred way to continue a statement.)
@	Indicates the assignable account; see the *RSTS/E System User's Guide*.
( )	Specify how operations are to be performed in expressions; see Section 8.4.5. Parentheses also enclose function arguments (see Chapter 9) or a project-programmer number (see the *RSTS/E System User's Guide*).
[ ]	Enclose a project-programmer number; see the *RSTS/E System User's Guide*.
+ – * / ^	Arithmetic operators; see Section 8.4.1.
=	Replacement operator; see Section 9.1. The equal sign also means "equal to" in numeric and string relational expressions; see Section 8.4.3.
= =	Means "approximately equal to" in numeric relational expressions; see Section 8.4.3.
<	Relational "less than" operator; see Section 8.4.3.
>	Relational "greater than" operator; see Section 8.4.3.
< >	Relational "not equal to" operator; see Section 8.4.3.

Table D–2: ASCII Character Codes

ASCII		Character	Remarks
Decimal	Octal		
0	000	NUL	Null, FILL character
1	001	SOH	CTRL/A
2	002	STX	CTRL/B
3	003	ETX	CTRL/C
4	004	EOT	End of transmission, CTRL/D
5	005	ENQ	CTRL/E
6	006	ACK	CTRL/F
7	007	BEL	Bell, CTRL/G
8	010	BS	Backspace, CTRL/H
9	011	HT	Horizontal tab, CTRL/I
10	012	LF	Line feed, CTRL/J
11	013	VT	Vertical tab, CTRL/K
12	014	FF	Form feed, page, CTRL/L
13	015	CR	Carriage return, CTRL/M
14	016	SO	CTRL/N
15	017	SI	CTRL/O
16	020	DLE	CTRL/P
17	021	DC1	CTRL/Q*, XON
18	022	DC2	CTRL/R
19	023	DC3	CTRL/S**, XOFF
20	024	DC4	CTRL/T
21	025	NAK	CTRL/U
22	026	SYN	CTRL/V
23	027	ETB	CTRL/W
24	030	CAN	CTRL/X
25	031	EM	CTRL/Y
26	032	SUB	CTRL/Z, end of file
27	033	ESC	Escape***
28	034	FS	
29	035	GS	
30	036	RS	
31	037	US	
32	040	SP	Space or blank
33	041	!	Exclamation mark
34	042	"	Quotation mark
35	043	#	Number sign
36	044	$	Dollar sign
37	045	%	Percent sign
38	046	&	Ampersand
39	047	'	Apostrophe
40	050	(	Left parenthesis
41	051	)	Right parenthesis
42	052	*	Asterisk
43	053	+	Plus sign
44	054	,	Comma
45	055	–	Minus sign or hyphen

(continued on next page)

Table D–2: ASCII Character Codes (Cont.)

ASCII		Character	Remarks
Decimal	Octal		
46	056	.	Period or decimal point
47	057	/	Slash
48	060	0	Zero
49	061	1	One
50	062	2	Two
51	063	3	Three
52	064	4	Four
53	065	5	Five
54	066	6	Six
55	067	7	Seven
56	070	8	Eight
57	071	9	Nine
58	072	:	Colon
59	073	;	Semicolon
60	074	<	Left angle bracket, "less than" sign
61	075	=	Equal sign
62	076	>	Right angle bracket, "greater than" sign
63	077	?	Question mark
64	100	@	At sign
65	101	A	Uppercase A
66	102	B	Uppercase B
67	103	C	Uppercase C
68	104	D	Uppercase D
69	105	E	Uppercase E
70	106	F	Uppercase F
71	107	G	Uppercase G
72	110	H	Uppercase H
73	111	I	Uppercase I
74	112	J	Uppercase J
75	113	K	Uppercase K
76	114	L	Uppercase L
77	115	M	Uppercase M
78	116	N	Uppercase N
79	117	O	Uppercase O
80	120	P	Uppercase P
81	121	Q	Uppercase Q
82	122	R	Uppercase R
83	123	S	Uppercase S
84	124	T	Uppercase T
85	125	U	Uppercase U
86	126	V	Uppercase V
87	127	W	Uppercase W
88	130	X	Uppercase X
89	131	Y	Uppercase Y
90	132	Z	Uppercase Z
91	133	[	Left square bracket
92	134	\	Backslash
93	135	]	Right square bracket
94	136	^ or ↑	Circumflex, up arrow, caret
95	137	← or _	Underscore or back arrow

(continued on next page)

Table D–2: ASCII Character Codes (Cont.)

ASCII		Character	Remarks
Decimal	Octal		
96	140		Grave accent
97	141	a	Lowercase a
98	142	b	Lowercase b
99	143	c	Lowercase c
100	144	d	Lowercase d
101	145	e	Lowercase e
102	146	f	Lowercase f
103	147	g	Lowercase g
104	150	h	Lowercase h
105	151	i	Lowercase i
106	152	j	Lowercase j
107	153	k	Lowercase k
108	154	l	Lowercase l
109	155	m	Lowercase m
110	156	n	Lowercase n
111	157	o	Lowercase o
112	160	p	Lowercase p
113	161	q	Lowercase q
114	162	r	Lowercase r
115	163	s	Lowercase s
116	164	t	Lowercase t
117	165	u	Lowercase u
118	166	v	Lowercase v
119	167	w	Lowercase w
120	170	x	Lowercase x
121	171	y	Lowercase y
122	172	z	Lowercase z
123	173	{	Left brace
124	174	|	Vertical line
125	175	}	Right brace
126	176	~	Tilde
127	177	DEL,RUBOUT	Delete, rubout

* CTRL/Q, or XON, resumes output if the TTYSET STALL characteristic is set.
** CTRL/S, or XOFF, stops output if the TTYSET STALL characteristic is set.
*** ALTMODE (ASCII 125) or PREFIX (ASCII 126) keys, which appear on some terminals, are translated internally into ESCAPE.

Appendix E
Hints for BASIC–PLUS / BASIC–PLUS–2 Compatibility

The following list contains guidelines for writing BASIC–PLUS programs that are compatible with BASIC–PLUS–2. Most of the guidelines also appear in the manual, in the description of the topic they pertain to.

1. Write programs in EXTEND mode.
2. Use the ampersand/RETURN combination, not the LINE FEED key, for continuation lines. The ampersand should appear either one space after the last significant character on the line or after one tab on otherwise blank lines.
3. Delimit all string constants with matching single (') or double (") quotation marks.
4. Include semicolons between strings and other items in PRINT statements. For example, enter:

```
20 PRINT "Customer Address"; A$
```

   You cannot use implied semicolons in BASIC–PLUS–2. The following PRINT statement is not compatible:

```
20 PRINT "Customer Address" A$
```

5. Use only one prompting message in an INPUT statement. For example:

```
50 INPUT "Enter your name and account number",N$,A
```

   The following INPUT statement, which contains two prompting messages, is not compatible with BASIC–PLUS–2:

```
50 INPUT "Enter your name"; N$, "Enter your account number";A
```

6. Use DEF*, not DEF, for user-defined functions.

7. Do not use parentheses when you define or call functions with no arguments. Use:

```
30 DEF* FNA%
```

   Do not use:

```
30 DEF* FNA%()
```

8. Use the CCPOS function instead of the POS function. CCPOS performs the same function in BASIC–PLUS and BASIC–PLUS–2. POS performs a different function in BASIC–PLUS and BASIC–PLUS–2.

9. When you write CHAIN statements, always include the optional keyword LINE. Use the form:

   CHAIN <string> LINE <line number>

   Do not use the form:

   CHAIN <string> <line number>

10. When you use arrays, always place the DIM statement before the first reference to the array it defines. For virtual arrays, place the DIM statement before the OPEN statement.

11. To disable an error handling subroutine, use the statement:

```
100 ON ERROR GOTO 0
```

    Do not use:

```
100 ON ERROR GOTO
```

12. Always include a RESUME statement in an error handling subroutine.

13. Always include the number sign (#) when specifying a channel number. For example:

```
500 CLOSE #2%,#4%
```

14. In the PRINT–USING statement, do not use $$ and ** in the same format field.

15. Do not use the COUNT option, the BLOCK option, or the USING option in GET statements. You cannot use these options with GET in BASIC–PLUS–2.

Appendix F
Programming Hints

This appendix is a collection of programming hints for the advanced BASIC–PLUS user:

- Section F.1 tells how to optimize BASIC–PLUS programs to reduce storage space and execution time.
- Section F.2 describes how to reduce disk access time.
- Section F.3 describes how to manipulate strings efficiently.
- Section F.4 explains in detail how the CVT functions convert data between numeric and string format. You will need this information if you plan to read files with block I/O that were created using a different I/O method.
- Section F.5 shows you the algorithm that BASIC–PLUS uses to access data in virtual array files. You might find this information useful if you need to access a virtual array file using block I/O.

F.1 Optimizing BASIC–PLUS Programs

By optimizing your use of statements and variables, you can write programs that use less storage space and execute more quickly.

Most of the time, these two goals are mutually exclusive: you can save space at the expense of time or vice versa. You need to decide which approach best fits your needs. If you can optimize both space and time, the entire system and your program benefit.

F.1.1 Optimizing Statement Formats

When BASIC–PLUS translates a source program into executable code, it creates structures called "statement headers." Statement headers contain information that the system needs to execute the program.

BASIC–PLUS creates a statement header for each line number in a program. In addition, certain statements always produce a statement header, regardless of where in a line they occur. These statements are DATA, DEF, DEF*, DIM, FNEND, FOR, NEXT, WHILE and UNTIL.

You can save statement header space by placing multiple statements on a single program line. Statements that require a statement header should be first on a line where possible. When you write multi-statement lines, use a separate text line for each statement. Your program will be easier to read.

You can also save statement header space by placing comments on the same line as statements. A REM statement and a comment with its own line number each require a 12–byte statement header.

F.1.2 Using Variables Efficiently

Assigning a temporary variable sometimes saves array addressing space. Consider:

```
10 FOR I% = 1% TO N% &
   \ S = S + X(I%) &
   \ S2 = S2 + X(I%) * X(I%)
20 NEXT I%
```

You can decrease the number of bytes required for storage by assigning a simple variable T equal to the subscripted variable X(I%):

```
10 FOR I% = 1% TO N% &
   \ T = X(I%) &
   \ S = S + T &
   \ S2 = S2 + T * T
20 NEXT I%
```

Besides using less storage space, this example also executes faster than the first one because BASIC–PLUS does not have to recalculate array addresses.

Individual variable names are often more economical than arrays because they require less overhead. If you use arrays, always dimension them with a DIM statement.

For efficiency, calculate quantities once and reuse them. For example:

```
10 D = SQR(B^2.-4. * A * C)/2% * A &
   \ PRINT -B/2% * A + D; -B/2% * A - D
```

This line is more efficient than:

```
10 D = SQR(B^2.-4. * A * C)/2% * A &
   \ PRINT -B/2% * A + SQR(B^2.-4. * A * C)/2% * A; &
   \ PRINT -B/2% * A - SQR(B^2.-4. * A * C)/2% * A
```

Use intermediate variables only when necessary. For example:

```
20 A = B + C &
   \ D = A + E &
   \ F = D + G
```

Unless you plan to use A and D later in the program, condense this line to:

```
20 F = B + C + E + G
```

Integers use less storage space than floating-point numbers, and integer arithmetic is much faster than floating-point arithmetic. Thus, you should use integer variables and constants where possible. Be sure to include the % suffix.

Use integer variables for array subscripts. Also use integers in FOR–NEXT loops with STEP values that are whole numbers in the range –32767% to 32766%.

Because of the way BASIC–PLUS stores variables, using variables with the same names but different data types (for example, A, A% and A$) saves space. However, this use of variables makes a program difficult to read and maintain.

Because BASIC–PLUS uses integers for logical operations, integer variables do not have to be compared to zero explicitly. For example:

```
30 IF M% <> 0 THEN 80
```

BASIC–PLUS interprets a nonzero value as true; thus, you can write this statement as:

```
30 IF M% THEN 80
```

Combine this technique with the IF statement modifier for a statement that uses even less storage space:

```
30 GOTO 80 IF M%
```

F.1.3 Using Constants Efficiently

Avoid ambiguous constants. When you specify constants, make them explicitly floating-point or integer by including a period (.) or a percent sign (%). Use integer constants for whole numbers in the range –32767% to 32767%.

BASIC–PLUS optimizes commonly used constants. The integer constants 0% and 1% and certain floating-point constants produce fewer bytes of translated code than other constants.

Each reference to an integer constant produces three bytes of translated code. But each reference to the integer constant 0% or 1% produces only one byte of translated code.

Each reference to a two-word floating-point constant produces seven bytes of translated code; each reference to a four-word floating-point constant produces eleven bytes of translated code. However, each reference to the floating-point constants 0. or 1. produces only one byte of translated code. In addition, a reference to one of the following floating-point constants produces three bytes of translated code:

- Any power of two
- Any whole number from 2. to 256.
- A power of ten up to 1000.

Both positive and negative values are optimized in all cases.

F.1.4 Statement Modifiers

Implied FOR loops use less memory and execute faster than FOR–NEXT loops. Consider:

```
10 FOR I% = 1% TO 10% &
   \ R% = R^2% &
   \ NEXT I%
```

The FOR and NEXT statements each produce a statement header. A FOR statement modifier, which produces only one statement header, creates a more efficient loop:

```
10 R% = R%^2% FOR I% = 1% TO 10%
```

This implied FOR loop uses about 30% less memory than the FOR–NEXT loop.

Where possible, use the WHILE and UNTIL statement modifiers instead of loops and IF–THEN statements. Consider:

```
10 IF X% < L% THEN X% = X% * X% &
   \ GOTO 10
```

You can perform this operation more efficiently with the statement:

```
10 X% = X% * X% WHILE X% < L%
```

F.1.5 Optimizing Statement Structure

When using multiple IF statements, use IF–THEN–ELSE instead of IF–THEN. Consider:

```
100 IF X% = W% THEN 250
110 IF X% = A% THEN 300
120 IF X% = K% THEN 500
130 IF X% = L% THEN 600
```

The following IF–THEN–ELSE statement uses 35% less memory than the previous example:

```
100 IF X% = W% THEN 250 ELSE &
          IF X% = A% THEN 300 ELSE &
                IF X% = K% THEN 500 ELSE &
                      IF X% = L% THEN 600
```

To compare a variable to a continuous range of values, use the ON–GOTO statement instead of the IF statement. For example:

```
100 ON X% - 3% GOTO 250, 300, 500, 600
```

This statement is more efficient than the following IF–THEN–ELSE statement:

```
100 IF X% = 4% THEN 250 ELSE &
          IF X% = 5% THEN 300 ELSE &
                IF X% = 6% THEN 500 ELSE &
                      IF X% = 7% THEN 600
```

You can use a similar technique with strings. For example:

```
80 X% = ASCII(A$) - 64% &
   \ ON X% GOTO 100, 200, 300, 400
```

These statements are more efficient than:

```
80 IF A$ = "A" THEN GOTO 100 ELSE &
         IF A$ = "B" THEN GOTO 200 ELSE &
               IF A$ = "C" THEN GOTO 300 ELSE &
                     IF A$ = "D" THEN GOTO 400
```

Use the same method to test random string responses. For example, the following statement compares A$ with the letters X, K, B, and Y:

```
80 ON INSTR(1%,"XKBY",A$) + 1% GOTO 100, 200, 300, 400, 500
```

You can also save space by using subroutines instead of user-defined functions, but be sure to exit with RETURN (not GOTO) statements.

F.2 Decreasing Disk Access Time

This section describes how to set up files to reduce disk access time. Some of these methods may require the assistance of your system manager.

Open files at the beginning of a program and preextend them to their maximum size. In addition, preallocate scratch files and, when you are finished using them, close them instead of deleting them. You can then reuse them with OPEN FOR INPUT statements. These techniques save disk space, reduce fragmentation of the directory structure, and decrease access time.

Keep large, frequently used files on separate disks. When two files are often open at the same time, they should also be stored on separate disks. Keep production files and accounts separate from development and scratch files where possible. If you cannot keep development and scratch files on separate disks, keep them in separate accounts.

Optimize file cluster sizes to further reduce disk access time. See Chapter 14 of this manual, the *RSTS/E System User's Guide*, and the *RSTS/E Programming Manual* for more information. The *RSTS/E Programming Manual* also describes other methods to save access time.

F.3 Manipulating Strings Efficiently

The following three algorithms truncate trailing blanks from a data record. The first two user-defined functions input a string and return the same string without trailing blanks and carriage return/line feeds.

The slowest algorithm successively reassigns the argument until it ends with a nonblank character:

```
1000 DEF* FNT$(X$) &
     \X$ = LEFT(X$,LEN(X$)-1%) &
         WHILE RIGHT(X$,LEN(X$)) <= " " &
               AND LEN(X$)>0%
1010     FNT$ = X$
1020 FNEND
```

The following algorithm is much more efficient. It scans backwards until a nonblank character is found. Only one assignment is made.

```
200 DEF* FNW$(X$) &
    \GOTO 2010 IF MID(X$,X%,1%)>" " &
          FOR X% = LEN(X$) TO 0% STEP -1% &
          \ X% = 0%
2010      FNW$ = LEFT(X$,X%)
2020 FNEND
```

The most efficient algorithm uses the data buffer directly, avoiding the assignment caused by the user-defined function. L% is the record length.

```
3000 FOR K% = L% TO 1% STEP -1% &
     \FIELD #2%, K%-1% AS L$, 1% AS L$ &
     \IF L$>" " THEN &
         FIELD #2%, K% AS L$ &
     \GOTO 3020
3010 NEXT K% &
     \LSET L$ = ""
3020 ! DONE
```

F.4 Converting Numeric Data

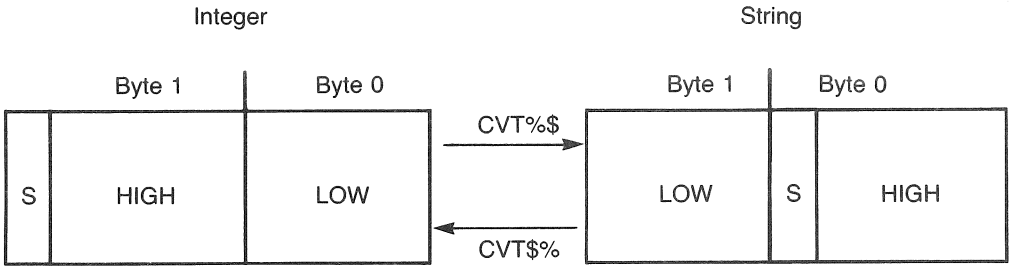
BASIC–PLUS stores numeric data in memory in either integer or floating-point format. When you use block I/O, you manipulate data in an intermediate I/O buffer. BASIC–PLUS lets you access the data in the buffer as

string data only. Thus, to process numeric data in your program, you must convert it from string to numeric format. After processing, you must convert the numeric data back to string data for output.

BASIC–PLUS provides four CVT functions to perform the necessary data conversions. These functions are implemented for speed instead of logical ordering. They use stack operations to convert data and thus reverse the expected ordering of bytes. The CVT$% and CVT%$ functions, which convert data between string and integer format, transpose the high and low order bytes of the word. The CVT$F and CVTF$ functions, which convert data between string and floating-point format, transpose the high and low order bytes of each word and also reverse the ordering of the words.

You are usually not aware of this reversal. When you manipulate data with block I/O, you use one CVT function when retrieving data from the I/O buffer and the related CVT function when loading data back into the I/O buffer for output. To illustrate, Figure F–1 shows the conversion of integer data by the CVT$% and CVT%$ functions.

Figure F–1: CVT Conversion of Integer Data



The CVT%$ function reverses the byte order of the integer data word. The CVT$% reverses the byte order of the string data, thus returning the integer to its correct byte order.

You do need to be aware of this reversal, however, when you use block I/O to read data not written by block I/O. In this situation, the data read into the buffer is in the correct byte order. The CVT$% function reverses the correct byte order of the data in the buffer. You must use the SWAP% function, which swaps the high and low order bytes of a word, to put the bytes back in the correct order.

For example, suppose that you need to read the date from a DOS magnetic tape label using non-file-structured block I/O. The system writes the standard PDP–11 internal representation of the date on a DOS magnetic tape label as a one-word integer value. To read the date using non-file-structured block I/O, you access it in two bytes of a buffer, which you define as a string variable (D$). You then move these two bytes into the integer variable D%, using the CVT$% function to convert the data from string to integer format. The CVT$% function reverses the order of the bytes, so you must use the SWAP% function to put them back in the correct order. For example:

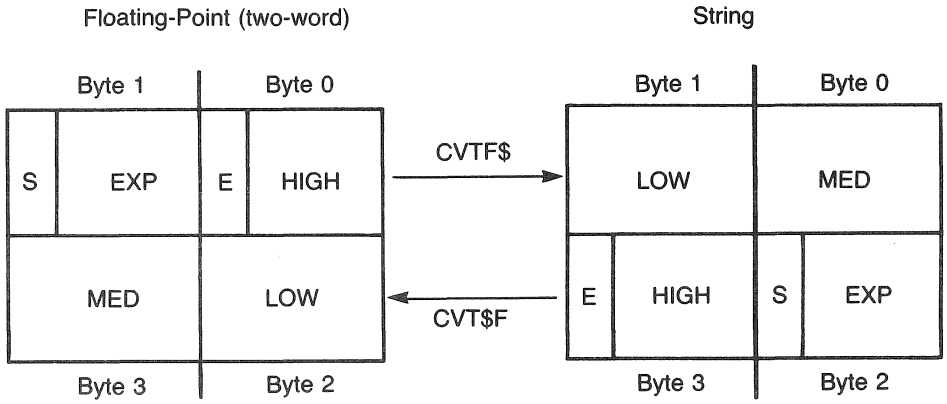
```
10 D% = SWAP%(CVT$%(D$))
```

You can then print the date with the DATE$ function:

```
20 PRINT DATE$(D%)
```

A more complex reversal occurs with floating-point data. Figure F–2 shows the conversion of two-word floating-point data by the CVTF$ and CVT$F functions.

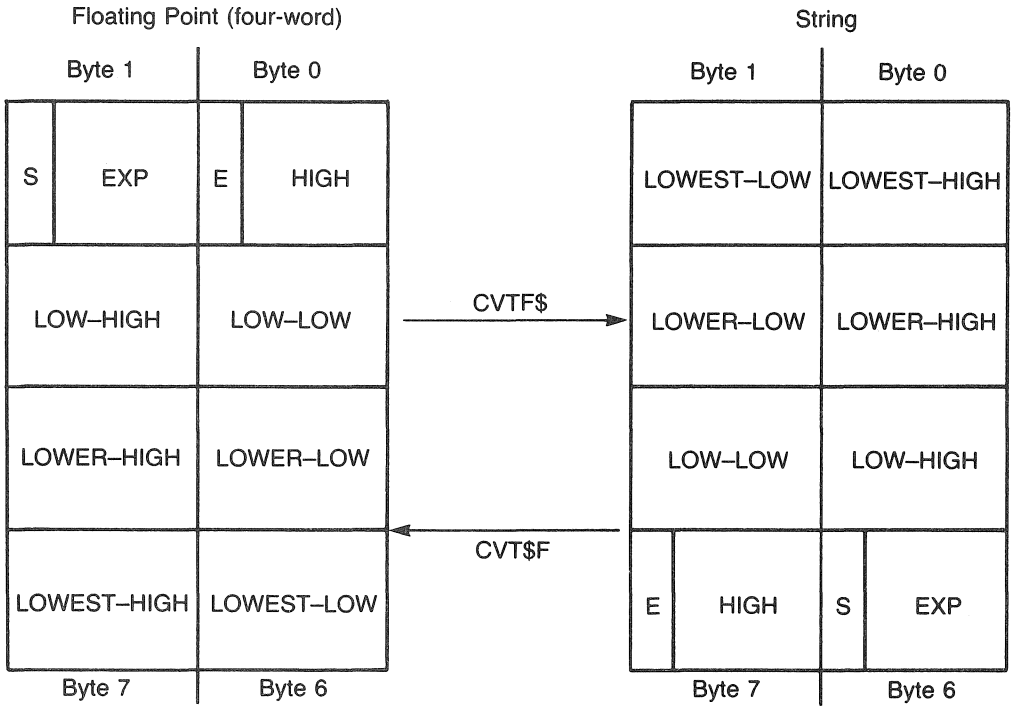
Figure F–2: CVT Conversion of Two-Word Floating-Point Data



When converting a two-word floating-point value to a four-byte string, the CVTF$ function stacks the bytes in the reverse of their original order. The sign and exponent bits are not in the standard format.

The same reversal occurs when the CVTF$ and CVT$F functions convert data between four-word floating-point and eight-byte string formats. Figure F–3 shows the results.

Figure F–3: CVT Conversion of Four-Word Floating-Point Data



As with integer data, you are not aware of this reversal when you manipulate floating-point data with block I/O. You convert data from string to floating-point format using the CVT$F function; you convert the floating-point data back to string format using the related CVTF$ function.

The reversal is evident, however, when you read data with block I/O that was not written using block I/O. Suppose, for example, that you read floating-point data from a virtual array file using block I/O methods. The virtual array processor does not convert data during input and output operations. Thus, data read into an I/O buffer from a virtual array file using block I/O statements is in correct byte order. But the CVT$F function, which you must use to convert the data in the buffer from string to floating-point format, reverses the correct byte order. Thus, the resulting numeric data is bad. (The same problem occurs when you read integer data from a virtual array file using block I/O.) You can solve this problem by reversing the byte order of the data in the buffer before converting it.

The following program creates a virtual array file that contains floating-point data. It then reads back the data using block I/O, reverses the byte order of the data in the buffer, and produces the floating-point representation using the CVT$F function.

```
100       OPEN 'VIRT.DAT' FOR OUTPUT AS FILE #1% &
          \ DIM #1, A(0) \ A,A(0) = RND * 1000.0 &
          \PRINT A &
          \ CLOSE 1% &
          \OPEN 'VIRT.DAT' FOR INPUT AS FILE #2%
220       L% = LEN(CVTF$(1.0)) &
          \ GET #2%
240       B$ = " &
          \ FOR I% = L% - 1% TO 0% STEP -1%
260                  FIELD #2%, I% AS B1$, 1% AS B1$
280                  B$ = B$ + B1$ &
          \ NEXT I%
300       PRINT CVT$F(B$);'SHOULD EQUAL';A
32767     CLOSE #1%,#2% &
          \ END
```

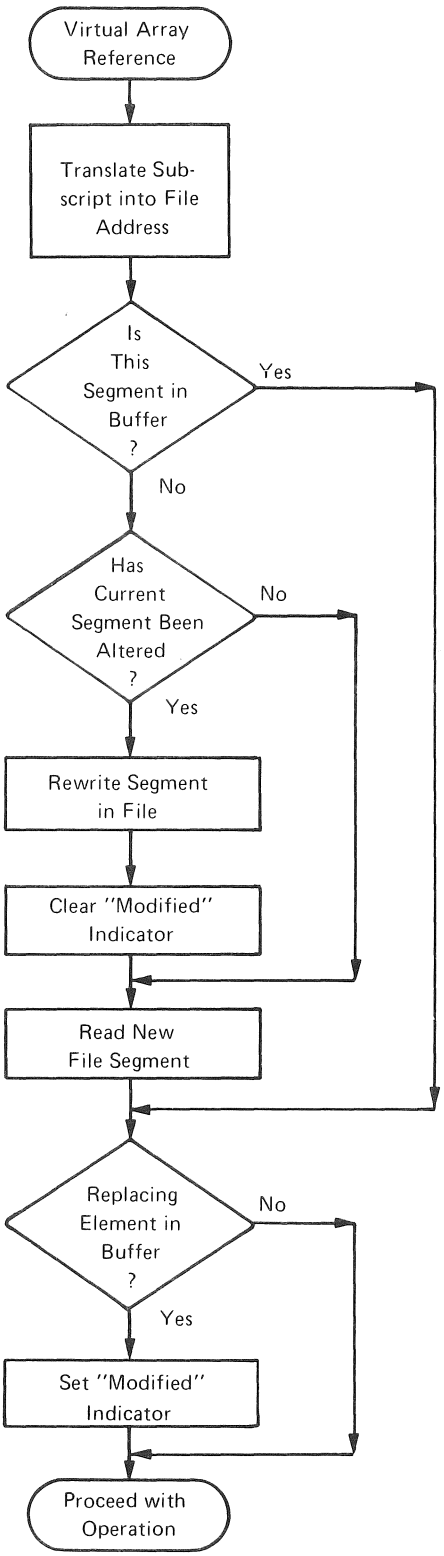
NOTE

Reading a virtual array file using block I/O is not recommended because you lose the advantage of the automatic virtual array addressing mechanism.

F.5 Accessing Algorithm for Virtual Arrays

Figure F–4 shows the algorithm that BASIC–PLUS uses to minimize the number of disk accesses needed in virtual array I/O. You might find this information useful if you need to access a virtual array file using block I/O.

Figure F–4: Virtual Array Accessing Algorithm



Glossary

You will find the terms in this glossary throughout this manual. They are defined here to help you if you are unfamiliar with computer terminology. The definitions of these terms come from a number of sources and are not intended to be absolute. Where possible, the most common industry usage has been the basis for defining a term.

Absolute value

The size of a quantity expressed without regard for its sign; the magnitude of a numeric quantity. For example, the numeric quantities +15 and –15 each have an absolute value of 15.

Alphanumeric

A contraction of alphabetic-numeric; the set of characters that consists of upper- and lowercase letters and the digits 0 through 9.

Argument

An independent variable whose value determines the value of a function or operation; the entity operated on by a command, function, or other instruction. For example:

```
SQR (X)
```

X is the argument of the SQR function. This function returns the square root of X.

Arithmetic operator

A symbol that represents one of the arithmetic operations, such as plus (+) for addition and minus (–) for subtraction.

Array

A series of items arranged in an orderly pattern; an ordered arrangement of elements in one or two dimensions.

Array dimension

One of the two possible array types; a list is a one-dimensional array, and a table is a two-dimensional array.

Array element

An item in an array.

ASCII code

An acronym for American Standard Code for Information Interchange; a standardized 7–bit code representing the 128 characters in which textual information is recorded.

Backslash

A character (\) used to separate statements when more than one statement appears on a program line.

Base 10

The decimal numbering system; indicates that there are ten symbols (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) allowed for each position in a decimal number; also Radix–10.

Base 2

The binary numbering system; indicates that there are two symbols (0 and 1) allowed for each position in a binary number; also Radix–2.

BASIC

Acronym for Beginner's All-purpose Symbolic Instruction Code; a computer language designed for direct communication between terminals (users) and a computer. BASIC is a registered trademark of Dartmouth College.

Binary

A base 2 numbering system; a condition or property that has two possibilities.

Binary digit

One of the two symbols (0 or 1) in the binary numbering system; see also Bit.

Binary number

The representation of a value as one or more binary digits; binary numbers increase in value by a factor of 2 for each position to the left (for example, the binary number 100100 equals the decimal number 36).

Bit

A contraction of Binary digit; the smallest unit of binary information.

Blank

One of the characters in a character set used to denote the presence of no information; a character used to print an empty space.

Branch

The transfer of control from the current instruction sequence within a program to an instruction other than the next sequential instruction.

Buffer

A temporary storage area used to contain data. Buffers hold data being passed between processes or devices that operate at different times or speeds.

Carriage return

A keyboard operation that causes the terminal print head to return to the left margin; usually combined with a line feed to send input to the computer from a terminal.

Central Processing Unit

That portion of a computer system that controls the interpretation and execution of instructions; also CPU or main frame.

Channel

A path between two or more separate units along which information can flow.

Character

One of a set of elementary symbols; a human-readable symbol that can be an upper- or lowercase letter from A to Z, a number from 0 to 9, or a special symbol; a machine-readable symbol consisting of a group of binary digits.

Character string

A sequence of characters treated as a single unit by the computer.

Circumflex

A character (^) used to represent exponentiation.

Code

A set of symbols and rules used to convert data from one representation to another; a system of correspondence between two units of information. The BASIC–PLUS language is a kind of code; so is the ASCII code used to store characters on the PDP–11 computer.

Colon

A character (:) used on some systems to separate statements when more than one statement appears on a line; see also Backslash.

Command

An order you give to a computer system that performs a predefined operation; the part of an instruction that specifies the operation to be performed.

Comment

Text that explains a particular program step but has no effect on program execution.

Compile

To save a translated BASIC–PLUS program in a disk file. (This term has a different meaning in other programming languages.)

Computer

A device capable of accepting information, processing that information, and providing a result; a device with self-contained memory that processes given information using prescribed operations and produces a result.

Concatenate

To unite in a series; to link together many items into one. In BASIC–PLUS, the plus sign (+) concatenates strings.

Conditional branch

A transfer of program control that takes place only when a prespecified condition is satisfied; see also Branch.

Constant

A quantity, value, or data representation that does not vary in value.

Continuation

A program state in which a program line is written across two or more terminal or text lines.

Control character

A special keyboard character that starts, changes, or stops an operation. For example, CTRL/O stops and restarts terminal output while a program is running.

Control key

A set of keyboard characters that causes a control action; a control key is usually the combination of the CTRL key and an alphabetic key, such as CTRL/C.

Counter

A device or storage location that accumulates numbers and allows their value to increase or decrease; a device used to represent the number of occurrences of a certain event.

CTRL/C

A combination of the keyboard characters CTRL and C that, when pressed simultaneously, halts program execution and returns you to command level.

Cursor

On a video terminal, a blinking white line or rectangle that marks the current position on the screen.

Data

Plural form of datum but commonly used as singular; any element of information that can be processed by a computer.

Data element

A series of data items within a related set of data.

Data item

A single data unit within a data element or data set.

Debugging

The process of detecting, locating, and correcting any mistakes in a computer program; the process of testing a program for errors.

Decimal number

A number in the base 10 numbering system. The base 10 numbering system is composed of ten possible symbols (0 through 9) with each number position representing that symbol times some power of 10.

Decimal point

The arithmetic symbol (.) appearing in decimal numbers that separates the whole and fractional parts of the number.

Default

An assumption made by a program when you do not provide a value.

Default keyboard monitor

The main keyboard you work in on a RSTS/E system. You enter the default keyboard monitor after you log in.

DELETE key

A terminal keyboard character that erases typed data from memory.

Delimiter

A character (such as a comma or semicolon) that separates the different parts of a statement. An example of the use of delimiters is:

```
10 PRINT "23*9=" , 23*9 ; 9*23
```

Device

A mechanical unit; a peripheral unit usually used to perform input and output operations.

Diagnostic

Pertaining to the detection and isolation of program errors or hardware malfunctions.

Dimension

The range or size defined for an array; a property whose number is used to uniquely determine the number of elements in a system of entities.

Directory

A device area that describes the layout of data on that device; a list of data contained in a file storage area in terms of names, length, and position.

Disk

A physical storage unit that stores data on rotating platters.

Dummy variable

An artificial value used to fulfill a condition without affecting an operation; a place holder. For example, the arguments you specify when you define a function with DEF* are dummy variables.

Editing

The process of modifying data, a program, or a file; the alteration of program format.

E format

Used in BASIC–PLUS to denote scientific notation. For instance, E+08 means "times ten to the eighth power" in this example:

8.867E+08 = 8.867*10^8 = 886700000.0

This notation is necessary in BASIC–PLUS, which can print only 1 to 6 or 15 digits (depending on the system).

Error

A discrepancy between a real quantity and a theoretically correct quantity; a deviation from true value; a mistake.

Error message

A notice from the computer that indicates an error and usually contains recovery information; a program message indicating the presence of a mistake.

Execution

The process of performing an instruction; the series of steps a computer performs to arrive at a result. The RUN command executes programs.

Exit

A method for stopping a program; the point of departure from a routine.

Exponent

Indicates the number of times a value is to be multiplied by itself. For example, in the expression 2^3, which means 2*2*2, 3 is the exponent.

Expression

Any legal combination of data and operators; a source language combination of one or more variables and operators.

Fatal error

A program error that causes a permanent exit from the current operations; an error that must be corrected before execution can proceed.

Field

One or more data elements treated as a unit. A field is usually part of a larger logical unit called a record.

File

An ordered collection of data capable of storage; a collection of related information; data to be transferred from executing programs to external (nonmemory) devices.

Format

The arrangement of data; the specified organization of information on a device, file, or printout.

Format error

A mistake that occurs in the specified organization of data.

Function

In mathematics, the relationship between a dependent variable and one or more independent variables. In BASIC–PLUS, a named group of instructions. BASIC–PLUS provides built-in functions (such as SQR and CVT$$) to help you manipulate numbers and strings. You can also define your own functions.

Hardware

The physical equipment or machinery that composes a computer system.

Header

The first part of a message; the initial portion of a message that acts as an identifying agent.

Increment

To add a quantity to another quantity; the quantity added.

Index

A number or quantity used to identify a particular item within a group of items. For example, the numeric value of a character is an index in the XLATE function.

Infinite loop

A repetitive instruction set with no means of exit; a loop that continues to repeat.

Initialize

To set up initial conditions. For example, you might set a variable to be incremented in a loop to an initial value.

Input

Information to be processed by a computer.

Instruction

A bit pattern that, when interpreted by the computer, directs it to perform an operation; a set of characters that defines an operation. You give instructions to BASIC–PLUS by entering commands and statements. BASIC–PLUS translates your commands and statements into a form that the computer can execute.

Integer

A whole number containing no fractional or decimal part, such as 100. In BASIC–PLUS, you indicate an integer with a % suffix, for example, 100%.

Interaction

A process of mutual communication between a human user and a computer system.

Interpreter

In BASIC–PLUS, the part of the BASIC–PLUS run-time system that executes translated code.

Job

The unit that RSTS/E uses to keep track of everything you do from the time you log in to the time you log out.

Job keyboard monitor

The keyboard monitor that manages a job. You change your job keyboard monitor to work in different RSTS/E command environments.

Keyboard

A device that encodes data by converting a pressed character key into an electrical signal.

Keyboard monitor

The part of a run-time system that you communicate with. Each RSTS/E keyboard monitor understands a set of commands.

Keyword

An essential word; a BASIC–PLUS verb that is a necessary element in the language.

Language

A systematic, unambiguous means for people to communicate with a computer; a set of representations, conventions, and associated rules used to convey information.

Language processor

Internal computer code that accepts data in one language and produces equivalent data in another language.

Line feed

The keyboard operation that shifts from one line position to the same horizontal position on the next vertical line.

Line number

The beginning number of a program line used for identification; a numeric label.

Log in

To gain access to a computer system.

Log out

To leave a computer system.

Logic error

A mistake in a program element that performs a decision-making function. For example, a GOTO statement that contains an incorrect line number is a logic error.

Loop

A sequence of instructions that is repeatedly executed until a terminating condition occurs.

Machine language

Binary instruction code that is directly readable by the computer.

Magnetic tape

A strip of material, usually plastic, that can store data in the form of magnetically polarized spots.

Magnitude

The absolute size or value of a number.

Main program

A computer program that controls all operations except those assigned to subprograms or subroutines.

Math function

A built-in routine designed to perform a mathematical computation.

Matrix

A general term for describing all the elements of a subscripted variable; an array.

Memory

A device on which data can be stored and from which it can be retrieved; internal computer storage.

Multi-statement line

A single program line composed of two or more statements distinguished by a statement separator. For example:

```
100 LET A = A+1 \ PRINT A
```

Negative step

A nonpositive loop increment index; a decremental loop. For example:

```
FOR I%=100% TO 1% STEP -2%
```

Nested loop

A loop embedded within another loop.

Nested parentheses

A parenthetical operation embedded within another parenthetical operation in an expression.

Nesting

The process of including a routine or block of data within another routine or block of data.

Nonprinting character

A character in the computer code set for which there is no corresponding graphic symbol. For example, the carriage return character (ASCII code 13 decimal) is a nonprinting character.

Null

The character with the ASCII code 000; an absence of information.

Null string

A string without content; an empty string.

Numeric variable

A variable that reserves a location in memory for a numeric value.

Octal

Pertaining to the base 8 numbering system.

One-dimensional array

An array composed only of rows; a list.

Operands

The data on which an operation is to be performed.

Operating system

Software that controls the execution of computer programs and performs system functions; an integrated collection of programs that manage computer operations.

Operation

The action specified by a single computer instruction.

Operator

The symbol or code that indicates the action to be performed; the portion of an instruction that tells the computer what to do. For example, in BASIC–PLUS the plus sign (+) is an operator that adds numbers and concatenates strings.

Output

Data that has been processed by the computer; the data produced as a result of transfer.

Overflow

A condition caused by a mathematical operation whose result exceeds the capacity of the computer; the portion of a result that cannot fit into a designated storage unit.

Paper tape

A strip of paper that can store data in the form of punched holes.

Parentheses

Symbolic language elements used to indicate nesting in an expression; a method of nesting where expression interpretation proceeds from the innermost to outermost level.

Peripheral device

Equipment that is separate or separable from the computer's processing unit; usually equipment that provides communication between the computer and its environment. Common peripheral devices are terminals, line printers, disks, and magnetic tapes.

Pointer

An instruction used to indicate the location of data.

Print

A process that causes the display or output of data.

Printer

A device that converts coded characters to human-readable hard copy.

Program

The complete sequence of instructions, data, and routines necessary for the solution of a problem.

Program line

A numbered group of data and instructions that, when combined with other lines, compose a program. A program line can contain one or more statements and can consist of one or more terminal or text lines.

Programming

The process of planning, writing, testing, and correcting the steps required for a computer to solve a problem.

Prompt

A feature of time-sharing systems that requests user input.

Punctuation

Special language characters such as commas and separators.

Quotation mark

A character (" or ') used in BASIC–PLUS to designate or delimit a string quantity.

Radian

A unit of plane angular measurement that is equal to the angle at the center of a circle subtended by an arc equal in length to the radius.

Random number

A number that is derived entirely by chance and is free from any bias toward predictable order.

Record

A group of data items treated as a unit. For example, a formatted ASCII record is a series of characters up to a line terminator (usually a carriage return/line feed), which ends the record.

Recursive

The repetitive process where the result of a cycle depends on the result of a previous cycle.

Reference

Data that indicates the location of information. For example, a variable name in a program is a reference to a memory location that stores a value.

Relational operator

A symbol or code that indicates the quantitative or qualitative relationship between two items of data.

Remark

See Comment.

RETURN key

A terminal key that causes typed data to be input to the computer; see also Carriage return.

Routine

A set of instructions and data that performs one or more specific operations; a subdivision of a computer program.

RUBOUT key

See DELETE key.

Run

An instruction that transfers a program from a file to memory and initiates execution; a single, continuous execution of a computer program or routine.

Run time

The time in which a program is executed; the actual amount of time required for a program to complete execution.

Run-time system

System software that manages part of the RSTS/E system. For example, the BASIC–PLUS run-time system manages the BASIC–PLUS programming environment.

Scientific notation

Multiplication of a number by a power of 10. For example, "7000" can be "70×10^2" or "7×10^3" in scientific notation. See also E format.

Sequential file

A storage unit in which data is available only in a consecutive sequence.

Sign

The symbol preceding a number that defines it as positive (+) or negative (–).

Software

All of the programs, procedures, rules, and peripheral information associated with the operation of a computer system.

Sort

The process of arranging items of information according to some portion of each item's content.

Source language

A language used by humans to write a computer program; the original form of a computer program before translation.

Square root

A factor of a number that, when squared, yields the number.

Statement

An expression or instruction written in a source language; an instruction to the computer to perform some operation.

Statement separator

A character or symbol that differentiates statements when two or more statements appear on the same program line.

Step

A single operation in a series of computer operations; to cause the execution of a single or specified number of operations. For example, the STEP value in a FOR loop increments or decrements the counter that determines how many times the loop will execute.

Storage

A general term for any device capable of retaining data; a device that receives, holds, and, at a later time, returns data; see also Memory.

Storage location

An identifiable area of memory that retains data.

String

A group of characters treated as a unit.

String variable

A variable that reserves a location in memory for character data. In BASIC–PLUS, you specify a string variable with a $ suffix, for example, A$.

Subroutine

A routine designed to be used by other routines to accomplish a specific task.

Subscript

A notation, written below and to the right of a set name, that represents a specified item in that set; an integer that identifies a particular item in an array. For example, in A(0%), the subscript 0% identifies the first element in the array A.

Subscripted variable

A variable name followed by one or more subscripts in parentheses; a notated variable that identifies the size of a storage location. For example, A(I%) is a subscripted variable.

Substring

One or more consecutive characters contained within a larger group of characters.

Syntax

The rules governing statement structure in a computer language; the structure of a language.

System

A combination of hardware and software that performs specific processing operations; a collection of components that forms a functional unit.

System commands

Instructions that you give the computer, such as RUN, which executes a program, and ASSIGN, which reserves an I/O device.

System manager

The person in charge of a computer system.

Table

A collection of data stored for easy reference; data stored in an array of rows and columns; see also two-dimensional array.

Terminal

A device, consisting of a keyboard and display mechanism, used to enter and receive data to and from a computer; any device that can send and receive information over a communication channel.

Terminal line

A horizontal line on the terminal screen. On video terminals, you can usually enter 80 or 132 characters on one terminal line.

Timesharing

A method of computer operation in which the system is shared by more than one user in a timeframe that appears to be simultaneous.

Translate

To convert source code into executable code. In BASIC–PLUS, translation occurs when you enter new statements, when you retrieve an existing source program with the OLD command, and when you run or chain to a source program.

Truncate

To reduce the size of a number by deleting one or more of the least significant digits; to drop digits from the end of a number, thus reducing precision.

Two-dimensional array

An array composed of columns and rows; a table.

Unconditional branch

An instruction that transfers program control to a specified location; an instruction that interrupts an operation and shifts control to another operation.

Underflow

A condition that occurs because the result of a mathematical operation is smaller than the program can handle; a situation in which a computed nonzero quantity is less than the smallest nonzero quantity that the computer can store.

Value

A quantity; the information represented by a data item.

Variable

An entity that can assume any of a given set of values; the symbolic representation of a storage location; a symbol whose value can change during a program.

Warning error

A mistake in a program that is not severe enough to halt program execution.

Warning message

A notice from the computer that a warning error is present in the program.

Index

A

ABS function, 9–28t
Account, assignable, D–2
ALT MODE key, 10–8
Alternate buffer I/O, 14–12, **17–10**
  example of, 17–10
  uses of, 17–10, 17–11
Ampersand/RETURN key, to continue line, 7–4, 7–5, 7–8
AND logical operator, 8–15
AND operation, diagram, 11–9
APPEND command, 5–8
  compared to DCL APPEND command, 5–9, B–1
Arithmetic
  expressions, 8–8
  floating-point, 11–13
  floating-point, accumulated error, 11–13
  integer, 11–4
  mixed-mode, 11–14 to 11–15
  with mixture of integer and floating-point data, 11–14 to 11–15
  operators, 8–9t
  scaled, 11–15 to 11–19
  string, 10–19
  string, precision, 10–21
  two's complement, 11–3
Arrays, 9–24 to 9–27
  determining size of, 9–26
  one-dimensional, 9–24
  saving addressing space for, F–2
  two-dimensional, 9–25f
  virtual, 16–1. *See also Virtual arrays*
ASCII
  character codes, D–3t
  function, 10–10t
ASSIGN command, B–1
  compared to DCL ASSIGN command, B–1
Assignable account, D–2
Assigning values to variables, 9–2
ATN function, 9–28t

B

.BAC file, C–15
  size of, C–16
.BAC program, 3–2
.BAS program, 3–1
BASIC–PLUS
  as default keyboard monitor, 2–3
  entering, 2–4
  leaving, 2–4
  run-time system, 2–2, C–16
  sample program, 7–2
BASIC–PLUS–2
  compatibility with, 9–29, E–1
  converting a program to, 6–14, 6–19
BASIC/BPLUS command, 2–4
Bit mask
  diagram of, 11–12
  example of, 11–12, 11–13
  uses of, 11–12
Bits
  masking, 11–12. *See also Bit mask*
  setting or clearing, 11–11
  testing, 11–11
  toggling (resetting), 11–10
Block I/O, 14–6, **17–1**
  accessing a specific block, 17–6
  accessing block of file over 65535 blocks, 17–7
  accessing I/O buffer, 17–11
  closing file, 17–4
  creating magnetic tape file, 17–19 to 17–20
  CVT functions in, 17–16
  defining I/O buffer, 17–12
  examples of, 17–18
  moving data into I/O buffer, 17–14
  opening file, 17–3
  printing line on terminal, 17–18
  processing numeric data, 17–16
  reading a block, 17–5
  reading and writing data, 17–5
  reading magnetic tape file, 17–19 to 17–20
  reading non-block I/O data, F–7
  reading virtual array data, F–9
  record blocking and deblocking, 17–20
  specifying number of characters to read, 17–8
  specifying number of characters to write, 17–7
  specifying offset into I/O buffer, 17–8
  storing numeric data in compact form, 17–19 to 17–20
  writing a block, 17–5

Block I/O file
  access to, 14–5
  closing, 14–15
  printing on line printer, 17–19
  record format, 14–5
BLOCK option, in GET and PUT statements, 17–6
Blocking record, definition, 17–20
BPCREF listing
  cross-reference table, 6–17
  example of, 6–17f
  global variable references, 6–19 to 6–20
  header line, 6–17
  local variable references, 6–19 to 6–20
  printing on line printer, 6–15
  statistical data, 6–18
  suspect line numbers and variables, 6–18
  variable reference codes, 6–18
BPCREF program, 6–14 to 6–22
  command format, 6–15
  command switches, 6–16t
  error messages, 6–21t
  output listing contents, 6–16 to 6–20
  running, 6–14
  sample cross-reference listing, 6–17f
Branch
  conditional, 9–13, 13–4, 13–5
  unconditional, 9–13
Buffer
  allowed sizes for each device, 14–11t
  default sizes for each device, 14–11t
  definition, 14–2
  determining size of, 17–3
  format of data in, 14–3
  setting size less than default, 14–12
  setting size of, 14–10
Buffer, I/O, 14–2. *See also Buffer*
BUFSIZ function, 17–3
BYE command, B–1

C

Card reader, record characteristics, 17–6
CATALOG command, 4–9 to 4–10
CCL, 2–2
  command, 2–2
  command, entering, 2–3
CCONT command, 6–8, 6–9
CCPOS function, 15–10
  with cursor control, 15–11
  use in program, 15–11
CHAIN statement, 13–22
  and CTRL/C, 13–23
  effect on I/O channels, 13–23, 14–15
  file operations, 13–23
  with privileged programs, 13–23
  with source and translated programs, 13–23
CHANGE statement, 10–3
Channel, I/O, 14–2. *See also I/O channel*
Channel 0, 14–3, 14–6
Channel number, 14–2, 14–6
Character data. *See String data*
Character set, BASIC–PLUS, 8–1, D–1
Characters, ASCII codes for, D–3t
Checksum, C–15
CHR$ function, **10–10t**, 15–3
CLOSE statement, 14–3, 14–15
  for block I/O file, 14–15
  for formatted ASCII file, 14–15
  with negative channel number, 14–15
  for virtual array, 14–15, 16–5
Cluster, disk, 14–12
Cluster size
  negative, 14–13
  optimal size for file, 14–12
  pack, 14–12
  UFD, 14–13
CLUSTERSIZE option, in OPEN statement, 14–12
Code
  ASCII character, D–3t
  source, 3–1
  translated, 3–2
Command
  APPEND, 5–8
  ASSIGN, B–1
  BYE, B–1
  CATALOG, 4–9 to 4–10
  CCL, 2–2, 2–3
  CCONT, 6–8, 6–9
  COMPILE, **4–8**, C–15
  CONT, 6–8, 6–9
  DCL, 2–2, 2–3
  DEASSIGN, B–2
  DELETE, 5–2 to 5–3
  EXTEND, 4–12, 7–9
  HELLO, B–2
  KEY, B–2
  keyboard monitor, 2–2
  LENGTH, 4–10
  LIST, 1–4, **4–5**

Command (Cont.)
LISTNH, 4–5
NEW, 1–3, **4–1** to **4–3**
NOEXTEND, 4–12
OLD, 1–5, **4–4**, C–15
REASSIGN, B–3
RENAME, 5–3
REPLACE, 1–6, **5–6**
RUN, 1–4, **4–6**
RUNNH, 4–6
SAVE, 1–5, **4–7**
SCALE, 4–11, **11–17**
summary, 3–2t
TAPE, B–4
UNSAVE, 5–6 to 5–7
Commands, summary of, B–1
Comments, 7–7
continuing with ampersand/RETURN, 7–8
continuing with LINE FEED key, 7–8
optimizing use of, F–2
Common memory, 13–23
COMP% string arithmetic function, 10–21t
Compatibility with BASIC–PLUS–2, hints for, E–1
COMPILE command, **4–8**, C–15
Compiled program. *See Translated program*
Concatenation, string, 8–10, 10–10
Concise Command Language, 2–2. *See also CCL*
Conditional
branch, 9–13, 13–4, 13–5
termination of FOR loop, 13–7
transfer to subroutine, 13–4
Constants, 8–3
ambiguous, 11–14, F–3
BASIC–PLUS optimization of, F–3 to F–4
bytes of translated code for, F–3 to F–4
integer, 8–4, 11–4
optimizing use of, F–3
real (floating-point), 8–3
string, 8–4, 10–1
CONT command, 6–8, 6–9
Control keys
CTRL/C, 6–12
CTRL/L, 10–8
CTRL/O, 6–13
CTRL/Q, 6–13
CTRL/R, 4–2
CTRL/S, 6–13
CTRL/U, 4–2, **5–2**
CTRL/Z, 14–4
summary of, B–5t
Control variable, in loop, 9–17
Conversion
ASCII to string, 10–5
integer array to string, 10–5
string to ASCII, 10–4
string to integer array, 10–4
Core common, 13–23
COS function, 9–28t
COUNT option
for disk, magnetic tape, and DECtape, 17–8
in GET statement, 17–8
in PUT statement, 17–7
for terminal, paper tape, and card reader, 17–8
CTRL/C, 6–5, 6–12
effect on I/O channels, 6–6, 6–8
effect on program variables, 6–6, 6–8
CTRL/C state. *See Ready state*
CTRL/L, 10–8
CTRL/O, 6–13
CTRL/Q, 6–13
CTRL/R, 4–2
CTRL/S, 6–13
CTRL/U, 4–2, **5–2**
CTRL/Z
in formatted ASCII file, 14–4, 15–14
with INPUT LINE statement, 15–16
Current program, 3–2
CVT functions, 17–16
conversion of integer data, F–7f
examples of, 17–17, 17–19 to 17–20
in LSET, RSET, and LET statements, 17–17
with SWAP%, F–7
CVT$$ function, **10–12** to **10–16**
bit mask in, 11–13
with INPUT LINE statement, 15–16
CVT$% function, 17–16t
with file name string scan, 11–12
CVT$F function, 17–16t
CVT%$ function, 17–16t
CVTF$ function, 17–16t

D

Data
block I/O, 14–5
constant, 8–3
file, 14–1, 14–2
floating-point, 8–2, **11–13.** *See also Floating-point data*
floating-point, conversion with CVT functions, F–8f
floating-point (in virtual array), 16–6
formatted (stream) ASCII, 14–4

Data (Cont.)
integer, 8–2, 11–1. *See also Integer*
integer, conversion with CVT functions, F–7f
integer (in virtual array), 16–6
printing on line printer, 17–19
printing on terminal, 9–3, 15–2, 17–18
printing to file or device, 15–3
reading from formatted ASCII file, 15–13
reading from terminal, 15–11
real, 8–2
string, 8–2, 10–1
string (in virtual array), 16–2, 16–6
supplying to running program, 9–7, 15–11
tailoring output format of, 15–4
variable, 8–5, A–1
virtual array, 14–4
Data file, 14–1, 14–2
DATA statement, 9–9 to 9–11, 10–6
Data types, 8–3t
DATE$ string function, 10–11t, 13–18t
DCL
keyboard monitor, 2–2
run-time system, 2–2
DCL commands, 2–2
APPEND, compared to BASIC–PLUS APPEND command, 5–9, B–1
ASSIGN, compared to BASIC–PLUS ASSIGN command, B–1
BASIC/BPLUS, 2–4
DEASSIGN, compared to BASIC–PLUS DEASSIGN command, B–2
DELETE, compared to BASIC–PLUS DELETE command, 5–3, B–2
entering, 2–3
RENAME, compared to BASIC–PLUS RENAME command, 5–4, B–3
DEASSIGN command, B–2
compared to DCL DEASSIGN command, B–2
Deblocking record, definition, 17–20
Debugging a program, 6–8 to 6–13
example of, 6–10
DECtape
as file-structured device, 14–10
buffer size, 14–11
OPEN statement for, 14–8
record characteristics, 17–6
DEF* statement, 9–31
arguments, 9–31, 9–33
multi-variable, 9–34
DEF* statement (Cont.)
multiple-line, 13–1
for string functions, 10–18
Default keyboard monitor, 2–2
DELETE command, 5–2 to 5–3
compared to DCL DELETE command, 5–3, B–2
DELETE key, 1–4, 4–2, **5–2**
Delimiters
carriage return/line feed, 10–8, 14–4
ESCAPE, 10–8, 14–4
form feed, 10–8, 14–4
in formatted (stream) ASCII file, 14–4
for INPUT LINE statement, 10–8
line feed, 10–8, 14–4
vertical tab, 14–4
DET matrix function, 12–8
Devices
file-structured, 14–2, 14–10
non-file-structured, 14–2, 14–10
record characteristics, 17–6t
DIF$ string arithmetic function, 10–20t
Difference, logical, 11–10
DIGITAL Command Language, 2–2. *See also DCL*
DIM statement, 9–26, 12–1
virtual array, 16–2
Directory listing, displaying, 4–9 to 4–10
Disk
as file-structured device, 14–10
buffer size, 14–11
default RECORDSIZE in block I/O, 17–6
opening for non-file-structured processing, 14–10
record characteristics, 17–6
Disk access time, decreasing, F–5
Disk file
extending, 17–9
locking and unlocking blocks in, 17–21
multi-user access to, 17–21
preextending to more than 65535 blocks, 14–14
preextending with FILESIZE option, 14–14
Diskette, buffer size, 14–11
Division with integers, 11–4
DMC11/DMR11, buffer size, 14–11
Dummy variables, 9–12

E

E format, 8–3
END statement, 6–6, 6–8, 9–37
effect on I/O channels, 6–6, 6–8

END statement (Cont.)
effect on program variables, 6–6, 6–8
explicit, 6–6
implicit, 6–6
EQV logical operator, 8–15
ERL variable, 6–6, 6–8, 13–17
ERR variable, 6–6, 6–8, 13–15, C–2
Error, untrapped
definition, 6–5
effect on I/O channels, 6–6, 6–8
effect on program variables, 6–6, 6–8
Error handling subroutine, 13–14 to 13–18
disabling, 13–16
transfer of control to multi-statement line, 13–16
Errors
abbreviations in descriptions, C–2t
BASIC–PLUS, C–1
checksum, C–15
disk, C–15
fatal, C–2
information, C–2
multiple meanings for, C–3
nonrecoverable, C–10t to C–14t
nontrappable, C–3t
?Program lost-sorry message, C–14
RSTS/E, C–1
severity, C–2t
trapping, C–1, C–2
user-recoverable, C–4t to C–10t
warning, C–2
ESCAPE key, 10–8
Exclusive OR logical operator, 8–15
Exclusive OR operation, diagram, 11–10
Executable program
file size, 4–9
protection code, 4–9
Execution
halting with CTRL/C, 6–5, 6–12
halting with STOP statement, 6–5, 6–8, 9–38
suspending with SLEEP statement, 13–20
EXP function, 9–28t
Exponential format, 8–3
Expressions, 7–3, **8–7 to 8–17**
arithmetic, 8–8
logical, 8–14, 11–6
logical, evaluation of, 11–7
numeric relational, 8–11
relational, 8–10
relational, evaluation of, 11–6
string, 8–10
string relational, 8–12
use of parentheses in, 8–16
EXTEND command, 4–12, 7–9
EXTEND mode, 4–12, **7–9**
EXTEND program format, 7–10f
comments, 7–12
line continuation, 7–12
spaces and tabs, 7–11
variable and function names, 7–11
EXTEND statement, 7–9

F

FIELD statement, 17–12
multiple definitions for same buffer, 17–14, 17–18
processing string with, 17–21
use in record blocking and deblocking, 17–20
use of subscripted string variables in, 17–13
virtual array strings in, 17–14
File
.BAC, 3–2, C–15
.BAC, size of, C–16
.BAS, 3–1
block I/O, 14–5
changing name, type, or protection code, 5–4, 14–16
closing, 14–3, 14–15
data, 14–1, 14–2
deleting from directory, 5–6, 14–17
extending a (disk), 17–9
formatted (stream) ASCII, 14–4, 15–3, 15–13
opening, 14–2, **14–6**
preextending to more than 65535 blocks, 14–14
preextending with FILESIZE option, 14–14
read and write access to, 14–8
reading and writing, 14–3
sharing, 17–21
virtual array, 14–4, 16–1. *See also Virtual arrays*
File name string scan, 11–12
File specification, RSTS/E, 4–4, 4–6, 4–8, 5–6
switches, 14–7
File-structured devices, 14–2, 14–10
FILESIZE option, in OPEN statement, 14–14
/FILESIZE switch, 14–14
FIX function, 9–28t
Floating-point
arithmetic, **11–13**
arithmetic, accumulated error, 11–13

Floating-point (Cont.)
constants, 8–3
variables, 8–6
Floating-point data, 8–2
assigned to integer, 11–4
conversion with CVT functions, F–8f
precision, 11–13
range of values, 11–13
in virtual array, 16–6
FOR loops
conditional termination of, 13–7
implied, 13–11, F–4
FOR statement, 9–17
FOR statement modifier, 13–11
Foreign buffers, 17–10. *See also Alternate buffer I/O*
Format
of data in buffer, 14–3
exponential, 8–3
program, 7–10
of statements, optimizing, F–1
Format field, 15–4. *See also PRINT–USING format field*
Formatted ASCII file, 15–3
access to, 14–4, 15–1
closing, 14–15
reading data from, 15–13, 15–14
record delimiters, 14–4, 15–14
record format, 14–4, 15–14
record length, 15–4, 15–14
writing data to read with INPUT statement, 15–3
writing nonprinting ASCII values, 15–3
Formatted ASCII I/O, 14–5, 15–1
examples, 15–17 to 15–19
Formatting characters, 15–2
in formatted ASCII file, 15–3
Functions
ABS, 9–28t
ASCII, 10–10t
ATN, 9–28t
BUFSIZ, 17–3
CCPOS, 15–10
CHR$, **10–10t**, 15–3
COMP%, 10–21t
COS, 9–28t
CVT, 17–16
CVT$$, **10–12**, 10–12t
CVT$%, 17–16t
CVT$F, 17–16t
CVT%$, 17–16t
CVTF$, 17–16t
Functions (Cont.)
DATE$, 10–11t, 13–18t
DET, 12–8
DIF$, 10–20t
EXP, 9–28t
FIX, 9–28t
INSTR, 10–10t
INT, 9–28t, 9–28
integer, user-defined, 11–5
INV, 12–8
LEFT, 10–9t
LEN, 10–10t
LOG, 9–28t
LOG10, 9–28t
MAGTAPE, 13–20
matrix, 12–8
MID, 10–10t
NUM$, 10–11t
NUM1$, 10–11t
PI, 9–28t
PLACE$, 10–21t
POS, 15–10
PRINT, 15–10
PROD$, 10–20t
QUO$, 10–20t
RAD$, 10–10t
recursive, 13–2
RIGHT, 10–10t
RND, 9–28t, 9–29
SGN, 9–27, 9–28t
SIN, 9–28t
SPACE$, 10–10t
SPEC%, 13–20, 17–21
SQR, 9–28t
string, 10–9
string, user-defined, 10–18
string arithmetic, 10–20t
string arithmetic, nesting, 10–24
STRING$, **10–12t**, 15–3
SUM$, 10–20t
summary of, A–2
SWAP%, 17–11, **17–17**
SYS, 11–11, 13–20
TAB, 15–10
TAN, 9–28t
TIME, 13–19t
TIME$, 10–11t, 13–19t
TIME$, format examples, 13–20t
TRN, 12–8
user-defined, 9–31
user-defined, multiple-line, 13–1
VAL, 10–11t
XLATE, 10–12t, **10–17**

G

GET statement, 17–5
GOSUB statement, 9–36
GOTO statement, 6–8, 9–13
  in debugging, 6–9

H

Header
  program, 4–5, 4–6
  statement, F–1. *See also Statement header*
HELLO command, B–2
Hints
  for BASIC–PLUS–2 compatibility, E–1
  programming, F–1

I

I/O
  alternate buffer, 14–12, **17–10**
  block, 17–1. *See also Block I/O*
  formatted ASCII, 14–5, 15–1
  formatted ASCII (examples), 15–17 to 15–19
  terminating, 14–15
  virtual array, 16–1. *See also Virtual arrays*
I/O buffer, 14–2. *See also Buffer*
I/O channel, 6–6, 14–6
  closing, 13–23, 14–15
  definition, 14–2
  determining buffer size of, 17–3
  effect of CHAIN on, 13–23
  opening terminal as, 15–15
I/O methods, 14–4 to 14–6
IF–GOTO statement, 9–13 to 9–15
IF statement, 9–13 to 9–15
IF statement modifier, 13–10
IF–THEN–ELSE statement, 13–5
IF–THEN statement, 9–13 to 9–15
Immediate mode
  debugging in, 6–8
  description, 1–2, **6–2** to **6–4**
  examples, 6–2
  limitations, 6–3
  variable assignments, 6–3
IMP logical operator, 8–15
Infinite loop, 6–5
INPUT LINE statement, 10–7, 15–15
  delimiters for, 10–8
  handling of carriage return character, 15–16
  record length for, 15–16
INPUT statement, 9–7, 10–7, **15–11** to **15–15**
  errors, 15–12
  reading data from file, 15–13
  reading data from terminal, 15–11
  record length for, 15–14
  use of prompting message in, 9–8, 15–11
INSTR string function, 10–10t
INT function, 9–28t, 9–28
Integer, 8–2
  arithmetic, 11–4
  as bit pattern, 11–2
  as logical value, 11–8
  as number, 11–1
  constants, 8–4, 11–4
  conversion with CVT functions, F–7f
  division, 11–4
  finding value from bit pattern, 11–3
  functions, user-defined, 11–5
  I/O, 11–5
  internal format, 11–2f
  in logical operations, 11–2, **11–6** to **11–13**
  storage, 11–1
  value range, 11–1
  variables, 8–6, 11–4
  in virtual array, 16–6
Integer function, INT, 9–28
Integers, optimizing use of, F–3
INV matrix function, 12–8

J

Job
  calling from SLEEP state, 13–21
  definition, 2–1
Job keyboard monitor, 2–2

K

KEY command, B–2
Keyboard
  buffer size, 14–11
  record characteristics, 17–6
Keyboard monitor, 2–1
  command, 2–2
  default, 2–2
  job, 2–2
  wait state. *See Ready state*
Keywords, 7–3, **8–1**
  list of reserved, A–20t
KILL statement, 5–7, 14–17

L

LEFT string function, 10–9t
LEN string function, 10–10t
LENGTH command, 4–10
LET statement, 9–2
  effect on string variables, 17–15
Line
  continuation, 7–4
  multi-statement, 7–4
  multi-statement, as spacesaving technique, F–2
  multi-statement, in immediate mode, 6–4
  program, 7–4
  terminal, 7–4
  text, 7–4
LINE FEED key, 7–5, 7–8, 10–8, 14–4
Line numbers, 7–1
Line printer, buffer size, 14–11
LINE variable, 6–6, 6–8, 6–13, 13–17
LIST command, 1–4, **4–5**
LISTNH command, 4–5
Literal, 15–9
Lock
  explicit, 17–21
  implicit, 17–21
  releasing with UNLOCK statement, 17–21
LOG function, 9–28t
LOG10 function, 9–28t
Logical
  difference, 11–10
  expressions, 8–14, 11–6
  expressions, evaluation of, 11–7
  operators, 8–15, 11–6
  product, 11–9
  sum, 11–10
  values. *See Logical values*
Logical operations
  how BASIC–PLUS performs, 11–8
  programming applications, 11–11
  truth values for, 8–15t, 11–9t
Logical operators
  AND, 8–15t
  EQV, 8–15t
  IMP, 8–15t
  NOT, 8–15t
  OR, 8–15t
  XOR, 8–15t
Logical values, 8–10
  –1% and 0% as, 11–6 to 11–7
  integers as, 11–8
  in relational expressions, 11–6
  storing, 11–7
  testing, 11–7
Loop index, 9–17
Loops, 9–15 to 9–24
  FOR, conditional termination of, 13–7
  FOR, implied, 13–11, F–4
  FOR–NEXT, 9–17
  nested, 9–20
  parts of, 9–17
  transferring control in and out of, 9–21
  types of, 9–17
  UNTIL–NEXT, 9–23
  WHILE–NEXT, 9–22
LSET statement, 17–14
  effect on string length, 17–14
  effect on string variables, 17–15

M

Magnetic tape
  as file-structured device, 14–10
  buffer size, 14–11
  OPEN statement for, 14–8
  record characteristics, 17–6
MAGTAPE function, 13–20
Mask, bit, 11–12. *See also Bit mask*
MAT initialization statement, 12–6
MAT INPUT statement, **12–4**, 15–16
MAT PRINT statement, **12–2**, 15–9
MAT READ statement, 12–2
MAT statements, 12–1
Mathematical functions, 9–27
Matrices, conforming, 12–8
Matrix
  calculations, 12–7
  determinant, 12–8
  functions, 12–8
  initializing, 12–6
  input from file with MAT INPUT, 15–16
  input from terminal with MAT INPUT, 12–4, 15–16
  inversion, 12–8
  manipulation, 12–1
  printing to file with MAT PRINT, 15–9
  printing to terminal with MAT PRINT, 12–2, 15–9
  reading with MAT READ, 12–2
  transposition, 12–8
MID string function, 10–10t
Mixed-mode arithmetic, 11–14 to 11–15
Mode
  read only, 14–8
  update, 14–8
MODE 1%, 17–21
MODE option, 14–8, 17–21
MODE option, in OPEN statement, 14–14

Modifiers
FOR statement, 13–11
IF statement, 13–10
multiple statement, 13–14
nested, 13–11
summary of statement, A–18
UNLESS statement, 13–11
UNTIL statement, 13–13
WHILE statement, 13–12
Multi-line statement, 7–5
Multi-statement line, 7–4
as spacesaving technique, F–2
and error-handling subroutine, 13–16
in immediate mode, 6–4

N

NAME–AS statement, **5–4**, 14–16
for nonprivileged users, 5–5
for privileged users, 5–5
Nesting
FOR–NEXT loops, 9–20
IF statements, 13–6
multiple-line DEF* functions, 13–3
string arithmetic functions, 10–24
subroutines, 9–36
techniques, 9–20f
NEW command, 1–3, **4–1** to **4–3**
NEXT statement, 9–18
NO SCROLL key, 6–14
NOEXTEND command, 4–12
NOEXTEND mode, 4–12, **7–9**
NOEXTEND program format, 7–10f
comments, 7–12
line continuation, 7–12
spaces and tabs, 7–11
variable and function names, 7–11
Non-file-structured devices, 14–2, 14–10
NOT logical operator, 8–15
Null device
in alternate buffer I/O, 17–11
buffer size, 14–11
use with FIELD statement, 17–21
Null string, 10–2
in INSTR function, 10–10
in relational expressions, 8–14
symbol for, 8–14
NUM variable, 12–5
NUM$ string function, 10–11t
NUM1$ string function, 10–11t
NUM2 variable, 12–5
Numbers, floating-point, 11–13. *See also Floating-point data*

O

OLD command, 1–5, **4–4**, C–15
ON ERROR GOTO statement, 6–5, 13–15, 15–12, C–1, C–2
ON–GOSUB statement, 13–4
ON–GOTO statement, 13–4
OPEN FOR INPUT, 14–7
OPEN FOR OUTPUT, 14–8
OPEN statement, 14–2, 14–3, **14–6** to **14–14**
creating file, 14–8
defaults in, 14–6
errors, 14–9t
for file-structured and non-file-structured devices, 14–6
forms of, 14–7
opening existing file, 14–7
options, 14–7, **14–10**
options, order of, 14–7
read/write access with, 14–8
for virtual array, 16–3
Operand, 7–3
Operators, 7–3
arithmetic, 8–9t
logical, 8–15, 11–6
numeric relational, 8–11t
precedence rules for, 8–16t
relational, 11–6
string relational, 8–12t, 10–3
summary of, A–2
Optimizing
constants, F–3
files, F–5 to F–6
integers, F–3
statement formats, F–1
statement structure, F–4 to F–5
strings, F–6
variables, F–2
Optional features of BASIC–PLUS, 1–1
Options
BLOCK, 17–6
CLUSTERSIZE, 14–12
COUNT, 17–7
FILESIZE, 14–14
MODE, 14–8, 14–14, 17–21
RECORD, 17–7
RECORDSIZE, 14–10
USING, 17–8
OR logical operator, 8–15
OR operation, diagram, 11–10

P

Pack cluster size, 14–12
Paper tape, record characteristics, 17–6
Paper-tape punch, buffer size for, 14–11
Paper-tape reader, buffer size for, 14–11
Parentheses, use in expressions, 8–16
Parity bit, 10–13
PI function, 9–28t
PLACE$ string arithmetic function, 10–21t
POS function, 15–10
Precision, string arithmetic, 10–21
  values in PROD$, QUO$, and PLACE$ functions, 10–24t
PREFIX key, 10–8
PRINT statement, 9–3, 10–9, **15–1** to **15–4**
  formatting characters in, 9–5, 9–6
  functions, 15–10
  output rules, 15–2
  printing numbers, 9–4, 15–2
  printing strings, 9–5, 15–2
  in scaled arithmetic, 11–17
  specifying output format, 9–5, 9–6, 15–2
  suppressing carriage return/line feed, 9–7, 15–2
PRINT–USING format field
  asterisk fill, 15–6
  comma and semicolon in, 15–8
  commas in numeric field, 15–7
  dollar sign, 15–7
  exclamation point, 15–4
  exponential format, 15–6
  insufficient numeric format, 15–7
  literal characters in, 15–8
  numeric field, 15–5
  numeric format too large, 15–8
  string field, 15–4, 15–5
  trailing minus sign, 15–6
PRINT–USING statement, 15–4 to 15–9
  printing numbers with, 15–5
  printing strings with, 15–4
  in scaled arithmetic, 11–17
Print zones
  in formatted ASCII file, 15–3
  printing data in, 9–5, 9–6, 15–2
  size of, 9–5, 15–2
Problems, software, reporting, C–16
PROD$ string arithmetic function, 10–20t
Product, logical, 11–9
Program
  .BAC, 3–2, C–15
  .BAS, 3–1
Program (Cont.)
  chaining to, 13–22
  changing file name, 5–4
  changing file type, 5–4
  changing name, 5–3
  changing protection code, 5–4
  changing scale factor, 11–18
  character set for, 8–1, D–1
  compiled. *See Program, translated*
  context, definition, 6–5
  continuing execution and detaching job with CCONT, 6–9
  continuing execution with CONT, 6–9
  continuing execution with GOTO, 6–9
  correcting syntax errors, 4–3
  correcting typing errors, 1–4, 4–2
  creating new, 1–3, 4–1
  creating with text editor, 4–3
  current, 3–2
  debugging, 6–8 to 6–13
  debugging, example of, 6–10
  deleting from disk storage, 5–6
  displaying length in memory, 4–10
  displaying on terminal, 1–4, **4–5**
  documenting, 7–6
  editing, 5–1
  entering statements, 4–2
  erasing lines, 5–2 to 5–3
  formatting, 7–4
  halting and checking execution, CTRL/C and PRINT LINE, 6–12
  halting execution with STOP, 6–8, 9–38
  header, 4–5, 4–6
  line, 7–4
  listing on terminal, 1–4, **4–5**
  in memory, 3–2
  modifying, 5–1
  optimizing, F–1. *See also Optimizing*
  parts of a, 7–1, 7–3
  printing on line printer, 4–8
  recompiling, C–15
  replacing saved, 1–6
  replacing saved source, 5–6
  retrieving source from disk, 1–5, 4–4
  running, 1–4, **4–6**
  running current, 4–6
  running saved, 4–6
  sample of, 7–2
  saving source on disk, 1–5, 4–7
  saving translated (executable) on disk, 4–8
  source, 3–1
  suspending execution, 13–20

Program (Cont.)
symbols in, D–1t
transferring control to, 13–22
translated, 3–2
translation of source code into executable code, 4–3
use of spaces and tabs in, 7–6
?Program lost-sorry error
causes, C–15
description, C–14
Programming hints, F–1
Programs, combining in memory, 5–8
/PROTECT switch, 4–9, 5–4, 14–7, 14–17
Protection code, 14–7, 14–8
changing, 5–4, 14–16
PUT statement, 17–5

Q

QUO$ string arithmetic function, 10–20t

R

RAD$ function, 10–10t
Random access, 17–6
Random number function, RND, 9–29
RANDOMIZE statement, 9–30
READ statement, 9–9 to 9–11, 10–6
Ready state
definition, 6–4
effect on privileged program, 6–6, 6–8
entering, 6–4, 6–5
possible actions in, 6–7
program status in, 6–6, 6–8t
Real data, 8–2. *See also Floating-point data*
REASSIGN command, B–3
Record blocking and deblocking
definition, 17–20
example, 17–20
use of FIELD statement in, 17–20
Record characteristics
for card reader, 17–6
for DECtape, 17–6
for disk, 17–6
for magnetic tape, 17–6
for paper tape, 17–6
for terminal, 17–6
RECORD option, in GET and PUT statements, 17–7
RECORDSIZE option, in OPEN statement, 14–10
RECOUNT variable, 17–8, **17–9**
Recursive function, 13–2
Relational
expressions, 8–10
expressions, evaluation of, 11–6
operators, 8–11, 8–12, 11–6
REM statement, 7–7
Remarks, 7–7
RENAME command, 5–3
compared to DCL RENAME command, 5–4, B–3
REPLACE command, 1–6, **5–6**
Reporting software problems, C–16
RESTORE statement, 9–12
RESUME statement, 13–15
transfer of control to multi-statement line, 13–16
RETURN key, 1–3, 10–8, 14–4
RETURN statement, 9–36
RIGHT string function, 10–10t
RND function, 9–28t, 9–29
/RONLY switch, 14–8
RSET statement, 17–14
effect on string length, 17–14
effect on string variables, 17–15
RSTS/E file specification, 4–4, 4–6, 4–8, 5–6
switches, 14–7
RUBOUT key, 4–2, 5–2
RUN command, 1–4, **4–6**
Run-time system
BASIC–PLUS, 2–2, C–16
definition, 2–1
RUNNH command, 4–6

S

SAVE command, 1–5, **4–7**
SCALE command, 4–11, **11–17**
Scale factor, 11–15
with .BAC files, 11–19
changing, 11–18
current, 4–11, 11–17
default value, 4–11, 11–15
displaying, 4–11, 11–18
effect on floating-point calculations, 11–16
with immediate mode statements, 11–19
pending, 4–11, 11–17
resetting to default value, 11–19
values for, 11–15
Scaled arithmetic, 11–15 to 11–19
SGN function, 9–27, 9–28t
Sign bit, 11–2
Sign function, SGN, 9–27
SIN function, 9–28t
SLEEP statement, 13–20

Software Performance Report (SPR), C–2
Software problems, reporting, C–16
Source program, 3–1
  retrieving from disk, 4–4
  saving on disk, 4–7
SPACE$ string function, 10–10t
SPEC% function, 13–20, 17–21
SPR, C–2
SQR function, 9–28t
Statement, definition, 7–3
Statement header
  definition, F–1
  saving space in, F–2
Statement modifiers, **13–10**
  efficient use of, 13–10, F–4
  summary of, A–18
Statements
  CHAIN, 13–22
  CHANGE, 10–3
  CLOSE, 14–15
  DATA, 9–9, 10–6
  DEF*, 9–31, 10–18
  DEF*, multiple-line, 13–1
  DIM, 9–26, 12–1
  DIM, virtual array, 16–2
  END, 6–8, 9–37
  ending, 1–3
  EXTEND, 7–9
  FIELD, 17–12
  FOR, 9–17
  GET, 17–5
  GOSUB, 9–36
  GOTO, 6–8, 6–9, 9–13
  IF–GOTO, 9–13
  IF–THEN, 9–13
  IF–THEN–ELSE, 13–5
  INPUT, 9–7, 10–7, **15–11** to **15–15**
  INPUT LINE, 10–7, 15–15
  KILL, 5–7, 14–17
  LET, 9–2
  LSET, 17–14
  MAT initialization, 12–6
  MAT INPUT, **12–4**, 15–16
  MAT PRINT, **12–2**, 15–9
  MAT READ, 12–2
  multi-line, 7–5
  NAME–AS, 5–4, 14–16
  NEXT, 9–18
  ON ERROR GOTO, 13–15, C–1, C–2
  ON–GOSUB, 13–4
  ON–GOTO, 13–4
  OPEN, 14–6
  optimizing format of, F–1
  PRINT, 9–3, 10–9, **15–1** to **15–4**
  PRINT–USING, 15–4 to 15–9
  PUT, 17–5
  RANDOMIZE, 9–30
  READ, 9–9, 10–6
  REM, 7–7
  RESTORE, 9–12
  RESUME, 13–15
  RETURN, 9–36
  RSET, 17–14
  SLEEP, 13–20
  STOP, 6–8, 9–38
  summary of, A–3 to A–18
  UNLOCK, 17–21
  UNTIL, 9–23
  WAIT, 13–21
  WHILE, 9–22
STATUS variable, 11–11, 14–8, 17–3
  bit tests, 17–4t
  with virtual arrays, 16–12
STEP expression, 9–18
STOP statement, 6–5, 6–8, 9–38
  effect on error-handling routines, 9–38
  effect on I/O channels, 6–6, 6–8
  effect on program variables, 6–6, 6–8
Stream ASCII file, 15–3
  access to, 14–4, 15–1
  closing, 14–15
  reading data from, 15–13, 15–14
  record delimiters, 14–4, 15–14
  record format, 14–4, 15–14
  record length, 15–4, 15–14
  writing data to read with INPUT statement, 15–3
  writing nonprinting ASCII values, 15–3
Stream ASCII I/O, examples, 15–17 to 15–19
String
  arithmetic, 10–19
  arithmetic, precision, 10–21
  ASCII conversions, 10–3
  concatenation, 8–10, 10–10
  constants, 8–4, 10–1
  conversion with CVT$$ function, 10–12
  data, 8–2, 10–1
  efficient use of, F–6
  expressions, 8–10, 8–12
  functions, 10–9
  functions, user-defined, 10–18
  input from file using INPUT LINE, 15–15
  input using INPUT, 10–7
  input using INPUT LINE, 10–7, 15–15
  input using READ and DATA, 10–6
  null, 10–2

String (Cont.)
output, 10–9
relational operators, 10–3
size, 8–5, 10–3
subscripted variables, 10–2, 17–13
translation with XLATE function, 10–17
variables, 8–7, 10–1
variables, effect of FIELD statement on, 17–12
variables, effect of LET statement on, 17–15
variables, effect of LSET and RSET statements on, 17–15
in virtual array, 16–2, 16–6
STRING$ string function, **10–12t**, 15–3
Subroutines, 9–35
calling, 9–36
conditional transfer to, 13–4
exiting, 9–36
nesting, 9–36
saving program space with, F–5
Subscripted variables, 9–24 to 9–27
in FIELD statement, 17–13
Sum, logical, 11–10
SUM$ string arithmetic function, 10–20t
Suspending program execution, 13–20
SWAP% function, **17–17**
in alternate buffer I/O, 17–11
with bit mask, 11–12
with CVT functions, F–7
effect on integer, 17–18
examples of, 17–18
uses of, 17–18
Switch
/FILESIZE, 14–14
/PROTECT, 4–9, 5–4, 14–7, 14–17
/RONLY, 14–8
in RSTS/E file specification, 14–7
SWITCH program, 2–4
Symbols, in programs, D–1t
Syntax error, correcting, 4–3
SYS function, 11–11, 13–20

T

TAB function, 15–10
TAB key, B–5
TAN function, 9–28t
TAPE command, B–4
TEMPnn.TMP file, 4–3
Terminal
buffer size, 14–11
opening as I/O channel, 15–15
Terminal (Cont.)
opening in special mode, 15–15
record characteristics, 17–6
Terminal keys, summary of, B–5t
Terminal line, 7–4
Terminal output
controlling, 6–13 to 6–14
stopping with CTRL/O, 6–13
suspending and resuming with CTRL/S and CTRL/Q, 6–13
suspending and resuming with NO SCROLL key, 6–14
Text line, 7–4
TIME function, 13–19t
TIME$ string function, 10–11t, 13–19t
format examples, 13–20t
Translated program, 3–2
file size, 4–9
protection code, 4–9
TRN matrix function, 12–8
Two's complement arithmetic, 11–3

U

UFD, 14–13
Unconditional branch, 9–13
UNLESS statement modifier, 13–11
UNLOCK statement, 17–21
UNSAVE command, 5–6 to 5–7
UNTIL statement, 9–23
UNTIL statement modifier, 13–13
Update mode, 14–8, 17–9, 17–21
and virtual arrays, 16–12
User-defined functions, 9–31
integer, 11–6
multiple-line, 13–1
string, 10–18
User File Directory, 14–13
USING option, in GET and PUT statements, 17–8

V

VAL string function, 10–11t
Values, logical. *See Logical values*
Variables, 8–5
assigning values to, 9–2
ERL, 6–6, 6–8, 13–17
ERR, 6–6, 6–8, 13–15, C–2
initial values for, 8–6
integer, 8–6, 11–4
LINE, 6–6, 6–8, 6–13, 13–17
naming, 8–6, A–1
NUM, 12–5

Variables (Cont.)
NUM2, 12–5
optimizing use of, F–2
real (floating-point), 8–6
RECOUNT, 17–8, **17–9**
STATUS, 11–11, 14–8, 17–3
string, 8–7, 10–1
string, effect of FIELD statement on, 17–12
string, effect of LET statement on, 17–15
string, effect of LSET and RSET statements on, 17–15
subscripted, 9–24 to 9–27, 17–13
summary of, A–1, A–2
temporary, F–2
Virtual array file
access to, 14–4
record format, 14–4
Virtual arrays, 14–5, **16–1**, 16–14
accessing algorithm, F–10f
buffer size, 16–4
changing values of elements in, 16–13
CLOSE with negative channel number, 16–5
closing file, 14–15, 16–5
compared with memory arrays, 16–1, 16–2
DIM statement, 16–2
effect of CHAIN statement on, 13–23
efficient referencing of, 16–7, 16–9, 16–14
Virtual arrays (Cont.)
examples of referencing, 16–7, 16–9, 16–14
examples of using, 16–12
file layout, 16–8f
initializing, 16–4
open on two I/O channels, 16–11
opening file, 16–3
preallocating size of, 16–11
preextending, 16–4
reading with block I/O, F–9
RECORDSIZE values for, 16–4
simultaneous access of, 16–12
storage on disk, 16–5
string length, 16–2, 16–6
string storage in, 16–2, 16–6
trailing null characters in, 16–3, 16–6
two-dimensional, 16–7

W

WAIT statement, 13–21
WHILE statement, 9–22
WHILE statement modifier, 13–12

X

XLATE string function, 10–12t, **10–17**
XOR logical operator, 8–15
XOR operation, diagram, 11–10

HOW TO ORDER ADDITIONAL DOCUMENTATION

DIRECT TELEPHONE ORDERS

In Continental USA and Puerto Rico call **800–258–1710**

In Canada call **800–267–6146**

In New Hampshire, Alaska or Hawaii call **603–884–6660**

DIRECT MAIL ORDERS (U.S. and Puerto Rico*)

DIGITAL EQUIPMENT CORPORATION
P.O. Box CS2008
Nashua, New Hampshire 03061

DIRECT MAIL ORDERS (Canada)

DIGITAL EQUIPMENT OF CANADA LTD.
940 Belfast Road
Ottawa, Ontario, Canada K1G 4C2
Attn: A&SG Business Manager

INTERNATIONAL

DIGITAL EQUIPMENT CORPORATION
A&SG Business Manager
c/o Digital's local subsidary
or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

*Any prepaid order from Puerto Rico must be placed with the Local Digital Subsidiary:
809–754–7575

Reader's Comments

Note: This form is for document comments only. Digital will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement. _____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number. _____

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN: Commercial Engineering Publications MK1-2/ H3
DIGITAL EQUIPMENT CORPORATION
CONTINENTAL BOULEVARD
MERRIMACK N.H. 03054

Do Not Tear - Fold Here and Tape

Cut Along Dotted Line

Printed in U.S.A.