BASIC–PLUS–2

User's Guide

Order Number: AA–JP35B–TK

May 1991

This manual provides tutorial information on BASIC–PLUS–2 language features. It also contains information on advanced program development techniques.

Revision/Update Information: This manual is a revision.

Operating System and Version: RSX–11M Version 4.6 or higher
RSX–11M–PLUS Version 4.3 or higher
Micro/RSX Version 4.3 or higher
RSTS/E Version 9.7 or higher

Software Version: BASIC–PLUS–2 Version 2.7

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 Equipment Corporation or its affiliated companies.

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

© Digital Equipment Corporation 1987, 1991.

All Rights Reserved.
Printed in U.S.A.

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation: BASIC–PLUS, BASIC–PLUS–2, DEC, DECnet, DECUS, Digital, DOCUMENT, Micro/RSX, PDP, PDP–11, RMS, RMS–11, RSTS, RSTS/E, RSX, RSX–11M, RSX–11M–PLUS, RX50, TK50, UNIBUS, VAX, VAXcluster, VAXinfo, VMS, and the Digital logo.

BASIC is a trademark of Dartmouth College.

This document was prepared using VAX DOCUMENT, Version 2.0.

Contents

Preface xix

Summary of Technical Changes xxv

1 Developing Programs in the BASIC Environment

1.1 Entering the Environment 1–1
1.2 Creating Programs 1–3
1.2.1 Creating a Program Interactively 1–3
1.2.2 Creating a Program Using a Text Editor 1–4
1.3 Running Programs 1–4
1.4 Indirect Command Files 1–5
1.5 Multiple-Unit Programs 1–6
1.6 Exiting from the Environment 1–7
1.7 Immediate Mode 1–8
1.8 Environment Commands 1–10
1.8.1 The $ System-Command 1–12
1.8.2 The APPEND Command 1–13
1.8.3 The BRLRES Command 1–13
1.8.4 The BUILD Command 1–13
1.8.5 The COMPILE Command 1–14
1.8.6 The DELETE Command 1–15
1.8.7 The DSKLIB Command 1–15
1.8.8 The EDIT Command 1–15
1.8.9 The EXIT Command 1–16
1.8.10 The EXTRACT Command 1–16
1.8.11 The HELP Command 1–17
1.8.12 The IDENTIFY Command 1–17
1.8.13 The INQUIRE Command 1–17
1.8.14 The LIBRARY Command 1–17
1.8.15 The LIST and LISTNH Commands 1–18
1.8.16 The LOAD Command 1–18

1.8.17 The LOCK Command . . . 1–19
1.8.18 The NEW Command . . . 1–19
1.8.19 The ODLRMS Command . . . 1–19
1.8.20 The OLD Command . . . 1–20
1.8.21 The RENAME Command . . . 1–20
1.8.22 The REPLACE Command . . . 1–20
1.8.23 The RUN and RUNNH Commands . . . 1–20
1.8.24 The SAVE Command . . . 1–21
1.8.25 The SCALE Command . . . 1–22
1.8.26 The SCRATCH Command . . . 1–22
1.8.27 The SEQUENCE Command . . . 1–22
1.8.28 The SET Command . . . 1–23
1.8.29 The SHOW Command . . . 1–23
1.8.30 The UNSAVE Command . . . 1–25
1.9 Setting Environment Defaults with an Initialization File . . . 1–26

2 Developing Programs at DCL Command Level

2.1 Using EDT to Create a BASIC–PLUS–2 Program . . . 2–1
2.2 Compiling a BASIC–PLUS–2 Program . . . 2–2
2.2.1 The BASIC Command . . . 2–3
2.2.2 BASIC Command Qualifiers . . . 2–3
2.2.3 Compiler Listings . . . 2–7
2.2.3.1 Source Program Listing . . . 2–7
2.2.3.2 Cross-Reference Listing . . . 2–9
2.2.3.3 Qualifier Summary . . . 2–11
2.3 Linking a BASIC–PLUS–2 Program . . . 2–12
2.3.1 The TKB Command . . . 2–12
2.3.2 The LINK Command . . . 2–13
2.4 Executing a BASIC–PLUS–2 Program . . . 2–14

3 Debugging Programs

3.1 Introduction . . . 3–1
3.2 Invoking the Debugger . . . 3–1
3.3 Sample Debugging Session in the BASIC–PLUS–2 Environment . . . 3–5
3.4 Sample Debugging Session at DCL Command Level . . . 3–7
3.5 Debugger Error Messages . . . 3–18

4 Program Elements

4.1 Line Numbers 4–1
4.2 Labels 4–2
4.3 Continuing Long Program Statements 4–2
4.4 Identifying Program Units 4–3
4.5 The BASIC–PLUS–2 Character Set 4–4
4.6 Program Documentation 4–4
4.7 Declarations and Data Types 4–5
4.7.1 Implicit Data Typing 4–6
4.7.2 Explicit Data Typing 4–7
4.8 Constants 4–7
4.9 Variables 4–9
4.9.1 Floating-Point Variables 4–9
4.9.2 Integer Variables 4–10
4.9.3 String Variables 4–10
4.9.4 Subscripted Variables 4–10
4.9.5 Initialization of Variables 4–11
4.10 Keywords and Reserved Words 4–11
4.11 Operands, Operators and Expressions 4–12
4.12 Assignment Statements 4–13

5 Simple Input-Output

5.1 Program Input 5–1
5.1.1 Providing Input Interactively 5–1
5.1.1.1 The INPUT Statement 5–1
5.1.1.2 The INPUT LINE and LINPUT Statements 5–4
5.1.1.3 Enabling and Disabling the Question Mark Prompt 5–4
5.1.2 Providing Input from the Source Program 5–5
5.1.2.1 The READ and DATA Statements 5–6
5.1.2.2 The RESTORE Statement 5–7
5.2 Program Output 5–8
5.2.1 Print Zones—the Comma and the Semicolon 5–9
5.2.2 Output Format for Numbers and Strings 5–12
5.3 Terminal-Format Files 5–14
5.3.1 Opening and Closing a Terminal-Format File 5–14
5.3.2 Writing Records to a Terminal-Format File 5–14

6 Control Statements

6.1 Statement Modifiers 6–1
6.1.1 The IF Modifier 6–2
6.1.2 The UNLESS Modifier 6–2
6.1.3 The FOR Modifier 6–2
6.1.4 The UNTIL Modifier 6–2
6.1.5 The WHILE Modifier 6–3
6.1.6 Nested Modifiers 6–3
6.2 Loops 6–3
6.2.1 FOR...NEXT Loops 6–4
6.2.2 WHILE...NEXT Loops 6–7
6.2.3 UNTIL...NEXT Loops 6–8
6.2.4 Nested Loops 6–9
6.3 Unconditional Branching 6–9
6.4 Conditional Branching 6–9
6.4.1 The ON...GOTO...OTHERWISE Statement 6–10
6.4.2 The IF...THEN...ELSE Statement 6–10
6.4.3 The SELECT...CASE Statement 6–13
6.5 The EXIT and ITERATE Statements 6–15
6.6 Executing Local Subroutines 6–16
6.6.1 The GOSUB and RETURN Statements 6–17
6.6.2 The ON...GOSUB...OTHERWISE Statement 6–18
6.7 Suspending and Halting Program Execution 6–19
6.7.1 The SLEEP Statement 6–19
6.7.2 The WAIT Statement 6–20
6.7.3 The STOP Statement 6–20
6.7.4 The END Statement 6–21

7 Declarations and Data Types

7.1 Declarative Statements 7–1
7.2 Data Types 7–1
7.3 Setting the Default Data Type and Size 7–4
7.4 Declaring Variables Explicitly 7–5
7.5 Declaring Named Constants Explicitly 7–6
7.5.1 Declaring Constants Within a Program Unit 7–7
7.5.2 Declaring Constants External to the Program Unit 7–7
7.6 Operations with Multiple Data Types 7–7
7.7 Allocating Static Storage 7–9
7.7.1 The COMMON Statement 7–9

7.7.2 The MAP Statement 7–10
7.7.2.1 Single Maps 7–10
7.7.2.2 Multiple Maps 7–12
7.7.3 FILL Items 7–13
7.7.4 Using COMMON and MAP in Subprograms 7–14
7.8 Dynamic Mapping 7–17

8 Functions

8.1 Built-In Functions 8–1
8.1.1 Using Numeric Functions 8–2
8.1.1.1 The ABS Function 8–2
8.1.1.2 The INT and FIX Functions 8–2
8.1.1.3 The SIN, COS, and TAN Functions 8–3
8.1.1.4 The LOG10 Function 8–4
8.1.1.5 The EXP Function 8–5
8.1.1.6 The RND Function 8–5
8.1.2 Using Data Conversion Functions 8–7
8.1.2.1 The ASCII Function 8–7
8.1.2.2 The CHR$ Function 8–7
8.1.3 Using String Numeric Functions 8–8
8.1.3.1 The FORMAT$ Function 8–8
8.1.3.2 The NUM$ and NUM1$ Functions 8–8
8.1.3.3 The VAL% and VAL Functions 8–10
8.1.4 Using String Arithmetic Functions 8–10
8.1.4.1 The PLACE$ Function 8–13
8.1.4.2 The PROD$ Function 8–14
8.1.5 Using Date and Time Functions 8–14
8.1.5.1 The DATE$ Function 8–14
8.1.5.2 The TIME$ Function 8–15
8.1.5.3 The TIME Function 8–15
8.1.6 Using Terminal Control Functions 8–16
8.1.6.1 The CTRLC and RCTRLC Functions 8–16
8.1.6.2 The ECHO and NOECHO Functions 8–17
8.2 User-Defined Functions 8–17
8.2.1 Single-Line DEF Functions 8–18
8.2.2 Multi-Line DEF Functions 8–19
8.3 External Functions 8–24
8.3.1 The FUNCTION, EXIT FUNCTION, and END FUNCTION Statements 8–24
8.3.2 The EXTERNAL Statement 8–25

9 String Handling

9.1	Introduction	9–1
9.2	Using Dynamic Strings	9–2
9.3	Using Fixed-Length Strings	9–4
9.4	String Virtual Arrays	9–5
9.5	Assigning String Data	9–6
9.5.1	The LET Statement	9–6
9.5.2	The LSET Statement	9–7
9.5.3	The RSET Statement	9–8
9.6	Manipulating String Data with String Functions	9–9
9.6.1	The LEN Function	9–9
9.6.2	The POS Function	9–10
9.6.3	The SEG$ Function	9–12
9.6.4	The MID$ Function	9–14
9.6.5	The STRING$ Function	9–15
9.6.6	The SPACE$ Function	9–15
9.6.7	The TRM$ Function	9–16
9.6.8	The EDIT$ Function	9–16
9.7	Manipulating String Data with Multiple Maps	9–18

10 Arrays

10.1	Introduction	10–1
10.2	Creating Arrays Explicitly	10–2
10.2.1	Creating Arrays with the DECLARE Statement	10–3
10.2.2	Creating Arrays with the DIM Statement	10–4
10.2.2.1	Declarative DIM Statements	10–5
10.2.2.2	Executable DIM Statements	10–5
10.2.3	Creating Arrays with the COMMON Statement	10–6
10.2.4	Creating Arrays with the MAP Statement	10–7
10.3	Creating Arrays Implicitly	10–7
10.3.1	Referencing an Undeclared Array Element	10–7
10.3.2	Using MAT Statements	10–9
10.3.2.1	The MAT Statement	10–10
10.3.2.2	The MAT READ Statement	10–12
10.3.2.3	The MAT INPUT [#] Statement	10–13
10.3.2.4	The MAT LINPUT [#] Statement	10–15
10.3.2.5	The MAT PRINT [#] Statement	10–16
10.3.2.6	Matrix I/O Functions	10–17
10.4	Array Input and Output	10–18
10.4.1	Assigning Values with the LET Statement	10–18
10.4.2	Listing Array Elements with the PRINT Statement	10–19

10.5 Matrix Operators 10–19
10.5.1 Arithmetic Matrix Operations 10–19
10.5.1.1 Assignment 10–20
10.5.1.2 Addition and Subtraction 10–20
10.5.1.3 Multiplication 10–20
10.5.2 Matrix Functions 10–21
10.5.2.1 The TRN Function 10–22
10.5.2.2 The INV Function 10–22
10.5.2.3 The DET Function 10–23

11 Program Segmentation

11.1 Introduction 11–1
11.2 BASIC–PLUS–2 Subprograms 11–2
11.2.1 SUB Subprograms 11–3
11.2.2 FUNCTION Subprograms 11–4
11.3 Declaring BASIC–PLUS–2 Subprograms 11–5
11.4 Accessing BASIC–PLUS–2 Subprograms 11–6
11.5 Passing Parameters to a BASIC–PLUS–2 Subprogram 11–8
11.6 Sharing Data Between Program Modules 11–12
11.6.1 Common Blocks and Maps 11–13
11.6.2 Files 11–17
11.7 Building Task Images 11–18
11.8 Non-BASIC–PLUS–2 Subprograms 11–24
11.8.1 Parameter-Passing Mechanisms 11–25
11.8.2 Declaring Non-BASIC–PLUS–2 Subprograms 11–27
11.8.3 Calling Non-BASIC–PLUS–2 Subprograms 11–29
11.9 MACRO Subprograms 11–30
11.9.1 Passing Parameters 11–30
11.9.2 Common Blocks and Maps 11–38
11.9.3 Initializing COMMONs and MAPs 11–41
11.9.4 Building Task Images 11–42
11.9.5 Handling Errors 11–45

12 File Input-Output

12.1 Introduction 12–1
12.2 Record Formats 12–2
12.2.1 Fixed-Length Records 12–2
12.2.2 Variable-Length Records 12–2
12.2.3 Stream Records 12–3
12.3 File Organizations 12–3
12.3.1 Terminal-Format Files 12–3

12.3.2 Sequential Files 12–3
12.3.3 Relative Files 12–4
12.3.4 Indexed Files 12–4
12.3.5 Virtual Files 12–5
12.4 Record Access and Record Context 12–5
12.5 I/O and Record Buffers 12–6
12.6 Accessing the Contents of a Record 12–7
12.6.1 The MAP Statement 12–7
12.6.2 The MAP DYNAMIC and REMAP Statements 12–7
12.6.3 The MOVE Statement 12–9
12.7 File and Record Operations 12–11
12.7.1 Opening Files 12–12
12.7.2 Creating Virtual Array Files 12–14
12.7.3 Locating Records 12–15
12.7.4 Reading Records 12–16
12.7.5 Writing Records 12–18
12.7.6 Deleting Records 12–20
12.7.7 Updating Records 12–21
12.7.8 Controlling Record Access 12–23
12.7.9 Accessing Records by Record File Address 12–24
12.7.10 Transferring Data to Terminal-Format Files 12–26
12.7.11 Resetting the File Position 12–26
12.7.12 Truncating Files 12–26
12.7.13 Renaming Files 12–27
12.7.14 Closing Files and Ending I/O 12–27
12.7.15 Deleting Files 12–28
12.8 File-Related Functions 12–28
12.8.1 The FSP$ Function 12–28
12.8.2 The FSS$ Function 12–29
12.8.3 The RECOUNT Function 12–36
12.8.4 The STATUS Function 12–37
12.9 OPEN Statement Clauses 12–37
12.9.1 The BUCKETSIZE Clause 12–37
12.9.2 The BUFFER Clause 12–39
12.9.3 The CLUSTERSIZE Clause 12–39
12.9.4 The CONNECT Clause 12–39
12.9.5 The CONTIGUOUS Clause 12–40
12.9.6 The DEFAULTNAME Clause 12–40
12.9.7 The EXTENDSIZE Clause 12–41
12.9.8 The FILESIZE Clause 12–41
12.9.9 The NOSPAN Clause 12–42
12.9.10 The RECORDTYPE Clause 12–42
12.9.11 The TEMPORARY Clause 12–43

12.9.12 The USEROPEN Clause 12–43
12.9.13 The WINDOWSIZE Clause 12–46

13 Formatting Output

13.1 Introduction 13–1
13.2 Using Format Strings 13–2
13.3 Printing Numbers 13–3
13.3.1 Specifying the Number of Digits 13–4
13.3.2 Specifying Decimal Point Location 13–5
13.3.3 Printing Numbers with Special Symbols 13–6
13.3.3.1 Commas 13–7
13.3.3.2 Asterisk Fill Fields 13–8
13.3.3.3 Currency Symbols 13–9
13.3.3.4 Negative Fields 13–10
13.3.3.5 E (Exponential) Format 13–10
13.3.3.6 Leading Zeros 13–11
13.3.3.7 Blank-If-Zero Fields 13–12
13.3.3.8 Debits and Credits 13–12
13.4 Printing Strings 13–12
13.4.1 Left-Justified Format 13–14
13.4.2 Right-Justified Format 13–14
13.4.3 Centered Fields 13–15
13.4.4 Extended Fields 13–15
13.5 Error Conditions 13–17

14 Compiler Directives

14.1 Introduction 14–1
14.2 Controlling the Compilation Listing 14–2
14.2.1 The %TITLE and %SBTTL Directives 14–2
14.2.2 The %IDENT Directive 14–4
14.2.3 The %PAGE Directive 14–4
14.2.4 The %LIST and %NOLIST Directives 14–5
14.2.5 The %CROSS and %NOCROSS Directives 14–6
14.3 Accessing External Source Files (%INCLUDE) 14–7
14.4 Controlling Compilation 14–8
14.4.1 Lexical Constants and Expressions (%LET) 14–9
14.4.2 The %VARIANT Directive 14–9
14.4.3 The %ABORT Directive 14–9
14.4.4 The %PRINT Directive 14–10
14.4.5 The %IF-%THEN-%ELSE-%END %IF Directive 14–10

15 Error Handling

15.1 Error Handlers 15–1
15.1.1 BASIC–PLUS–2 Default Error Handling 15–2
15.1.2 User-Written Error Handlers 15–2
15.2 Identifying Errors 15–4
15.2.1 Determining the Error Number (ERR) 15–4
15.2.2 Determining the Error Line Number (ERL) 15–5
15.2.3 Determining Where the Error Occurred (ERN$) 15–6
15.2.4 Determining the Error Message Text (ERT$) 15–6
15.2.5 Ctrl/C Trapping 15–7
15.3 Handling Errors in Multiple-Unit Programs 15–8
15.4 Returning to BASIC–PLUS–2 Error Handling 15–10
15.5 Leaving an Error Handler 15–11

16 Instruction and Data Space

16.1 Introduction 16–1
16.2 Building Tasks in Instruction and Data Space 16–1
16.2.1 MACRO Subprograms 16–3
16.2.2 Overlaid Tasks 16–4

17 Advanced Input-Output

17.1 Introduction 17–1
17.2 RMS I/O to Magnetic Tape 17–2
17.2.1 Allocating a Tape 17–2
17.2.2 Initializing a Tape on RSX–11M Systems 17–2
17.2.3 Initializing a Tape on RSX–11M–PLUS Systems 17–3
17.2.3.1 The MOUNT Command 17–3
17.2.3.2 The INITIALIZE Command 17–4
17.2.3.3 The DISMOUNT Command 17–4
17.2.3.4 Example of Initializing a Tape on RSX–11M–PLUS 17–4
17.2.4 Initializing a Tape on RSTS/E Systems 17–5
17.2.4.1 The INITIALIZE Command 17–5
17.2.4.2 The MOUNT Command 17–5
17.2.4.3 Example of Initializing a Tape on RSTS/E 17–6
17.2.5 Opening a File on Tape 17–7
17.2.5.1 ACCESS Clause 17–8
17.2.5.2 BLOCKSIZE Clause 17–8
17.2.5.3 MAP Clause 17–8
17.2.5.4 RECORDSIZE Clause 17–9
17.2.5.5 NOREWIND Clause 17–9

17.2.6 Writing Records to Tape 17–10
17.2.7 Adding New Records to Tape 17–11
17.2.8 Reading Records on Tape 17–12
17.2.9 Locating Records on Tape 17–14
17.2.10 Truncating Files on Tape 17–15
17.2.11 Dismounting a Tape 17–17
17.2.12 Closing a File on Tape 17–18
17.3 Device-Specific I/O 17–18
17.3.1 Device-Specific I/O to Unit Record Devices 17–19
17.3.2 Device-Specific I/O to Magnetic Tape Devices 17–19
17.3.2.1 Allocating and Mounting a Tape 17–20
17.3.2.2 Opening a File on Tape 17–20
17.3.2.3 Opening a Tape File for Output 17–22
17.3.2.4 Opening a Tape File for Input 17–22
17.3.2.5 Closing a File on Tape 17–23
17.3.2.6 Using the MAGTAPE Function 17–23
17.3.2.7 Writing Records to Tape 17–27
17.3.2.8 Adding New Records to Tape 17–29
17.3.2.9 Reading Records on Tape 17–30
17.3.2.10 Locating Records on Tape 17–31
17.3.2.11 Deleting Records on Tape 17–33
17.3.2.12 Dismounting a Tape 17–34
17.3.3 Device-Specific I/O to Disk Devices 17–34
17.3.3.1 Allocating and Mounting a Disk 17–35
17.3.3.2 Opening a File on Disk 17–35
17.3.3.3 Writing Records to Disk 17–36
17.3.3.4 Adding New Records to Disk 17–37
17.3.3.5 Reading Records on Disk 17–38
17.3.3.6 Locating Records on Disk 17–40
17.3.3.7 Deleting Records on Disk 17–40
17.3.3.8 Dismounting a Disk 17–41
17.4 Network I/O 17–41

18 Libraries

18.1 Introduction 18–1
18.2 BASIC–PLUS–2 Libraries 18–1
18.2.1 BASIC–PLUS–2 Memory-Resident Libraries 18–1
18.2.2 BASIC–PLUS–2 Object Module Libraries 18–3
18.3 User-Created Libraries 18–3
18.3.1 Creating a Memory-Resident Library 18–4
18.3.2 Selecting a Memory-Resident Library 18–4
18.3.3 Creating an Object Module Library 18–5

18.3.4 Selecting an Object Module Library 18–5
18.3.4.1 Selecting Both the Default and a User-Created Object Module Library 18–6
18.3.4.2 Selecting a User-Created Object Module Library 18–6
18.4 RMS–11 Libraries 18–7
18.4.1 The RMS–11 Memory-Resident Libraries 18–7
18.4.2 Selecting an RMS–11 Memory-Resident Library 18–8
18.4.3 The RMS–11 Object Module Library 18–9
18.4.4 RMS–11 ODL Files 18–9
18.4.5 Selecting an RMS–11 ODL File 18–10
18.5 Clustering Memory-Resident Libraries 18–11
18.6 Remote File Access 18–12

19 Utilities

19.1 The Optimizer Utility 19–1
19.1.1 Invoking the Optimizer Utility 19–1
19.1.1.1 The OPT Command 19–2
19.1.1.2 The RUN $BP2OPT Command 19–3
19.1.2 Choosing a Segment Size 19–4
19.1.3 Optimizer Error Messages 19–9
19.2 The Dump Analyzer Utility 19–12
19.3 The Resequencer Utility 19–13
19.3.1 Creating a Resequencer Command File 19–15
19.3.2 Formatting Commands in a Resequencer Command File 19–15
19.3.3 Resequencer Utility Error Messages 19–16

20 Optimization Techniques

20.1 Writing Transportable Programs 20–1
20.2 Optimizing Your Program 20–2
20.2.1 I/O Operations 20–2
20.2.2 Assigning Variables 20–3
20.2.3 Choosing Compiler Options 20–3
20.2.4 Selecting Data Types 20–3
20.2.5 Arithmetic Operations 20–4
20.2.6 Using Control Structures 20–4
20.2.7 Selecting Libraries 20–7
20.2.8 RMS–11 Operations 20–7
20.2.9 Static and Dynamic Storage 20–8
20.2.10 Extending Memory 20–9

A Compile-Time and Environment Error Messages

A.1 Diagnosing Compile-Time and Environment Errors A–1
A.2 Error Message Format A–2
A.3 Alphabetical List of Error Messages A–2

B Run-Time Error Messages

B.1 Diagnosing Run-Time Errors B–1
B.2 Error Message Format B–2
B.3 Numerical List of Error Messages B–2
B.4 Alphabetical List of Error Messages B–25
B.5 Non-BASIC Errors B–29

C ASCII Codes and Data Representation

C.1 Radix–50 Character Set C–6
C.2 BYTE Integer Format C–9
C.3 WORD Integer Format C–10
C.4 LONGWORD Integer Format C–10
C.5 Floating-Point Formats C–11
C.5.1 Single-Precision Format C–12
C.5.2 Double-Precision Format C–13
C.6 String and Array Formats C–14
C.6.1 Array Formats C–14
C.6.2 Array Descriptor Word 2 .... C–18

Index

Examples

16–1 Map File Illustrating Instruction and Data Space 16–2
19–1 Example of an Optimizer Listing File 19–5

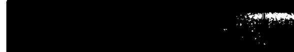
Figures

1–1	Running Multiple-Unit Programs	1–6
7–1	Multiple Maps	7–12
11–1	Tree Figure Representing the Overlay Structure	11–20
11–2	Example of Overlay Structure	11–21
11–3	Nonoverlay and Overlay Memory Requirements	11–22
11–4	Argument List Format	11–32
16–1	Task Layout	16–4
20–1	Comparison of Static Storage to Dynamic Storage	20–9
C–1	Byte-Length Integer Format	C–9
C–2	Word-Length Integer Format	C–10
C–3	Longword Integer Format	C–11
C–4	Floating-Point Format	C–12
C–5	Single-Precision Format	C–12
C–6	Double-Precision Format	C–13
C–7	Dynamic String Format	C–14
C–8	Format of Arrays in Memory	C–15
C–9	Format of Arrays in Virtual Memory	C–16
C–10	Dynamic Arrays	C–17
C–11	Dynamic String Array Pointers	C–18
C–12	Array Descriptor Word 2	C–19

Tables

1–1	Environment Commands	1–11
1–2	Commands Allowed in an Initialization File	1–27
3–1	Debugger Commands	3–3
4–1	Predefined Constants	4–8
7–1	BASIC–PLUS–2 Data Types	7–3
7–2	Result Data Types in Expressions	7–8
7–3	FILL Item Formats, Representations, and Default Allocations	7–13
8–1	String Arithmetic Functions	8–11
8–2	Precision of String Arithmetic Functions	8–11
9–1	String Modification	9–2
9–2	EDIT$ Options	9–16
10–1	MAT Statements	10–9

10–2	MAT Statement Keywords	10–10
11–1	BASIC–PLUS–2 Parameter-Passing Mechanisms	11–26
12–1	Record Context After a FIND Operation	12–16
12–2	Record Context After a GET Operation	12–17
12–3	Record Context After a PUT Operation	12–18
12–4	File Name String: Flag Word Bytes 1–30	12–30
12–5	File Name String: Scan Flag Word 1	12–31
12–6	File Name String: Scan Flag Word 2	12–32
13–1	Format Characters for Numeric Fields	13–7
13–2	Format Characters for String Fields	13–13
17–1	MODE Values	17–22
17–2	MAGTAPE Function Codes	17–24
17–3	Tape Status Word	17–27
18–1	ODL Files Supplied by RMS–11	18–9
19–1	Resequencer Commands	19–15
20–1	BASIC–PLUS–2 Substitutes for System Services	20–1
B–1	Alphabetical List of Run-Time Errors	B–25
C–1	ASCII Codes	C–2
C–2	Radix–50 Character Set	C–7
C–3	ASCII and Radix–50 Equivalents	C–8



Preface

Intended Audience

This manual provides tutorial information on BASIC–PLUS–2 language features. Readers are presumed to have some previous knowledge of BASIC or another high-level programming language. This manual should be used with the other manual in the documentation set.

Operating Systems and Versions

BASIC–PLUS–2 Version 2.7 runs on the following operating systems and versions:

RSX–11M Version 4.6 or higher
RSX–11M–PLUS Version 4.3 or higher
Micro/RSX Version 4.3 or higher
RSTS/E Version 9.7 or higher

Associated Documents

This manual is one of two manuals that form the BASIC–PLUS–2 document set. The other manual in the document set, the *BASIC–PLUS–2 Reference Manual*, provides reference material and examples on all BASIC–PLUS–2 commands, directives, statements, and functions. If you are unfamiliar with a topic, you may want to read the information contained in this manual before consulting the *BASIC–PLUS–2 Reference Manual.*

If you are an inexperienced BASIC programmer, you should read the following manuals before using the BASIC–PLUS–2 document set:

- *Introduction to BASIC*
- *BASIC for Beginners*
- *More BASIC for Beginners*

Document Structure

This manual has 20 chapters and three appendixes.

Chapter 1	Describes how to develop programs in the BASIC environment
Chapter 2	Describes how to develop programs from DCL command level
Chapter 3	Describes how to use BASIC–PLUS–2 debugger to debug programs
Chapter 4	Explains the fundamental elements of BASIC–PLUS–2 programs
Chapter 5	Explains simple input and output procedures
Chapter 6	Shows how to control the flow of program execution
Chapter 7	Explains data definitions
Chapter 8	Explains how to use functions
Chapter 9	Explains how to handle strings
Chapter 10	Shows how to use arrays
Chapter 11	Describes how to write modular programs and include calls to MACRO subprograms
Chapter 12	Explains how to manage RMS files
Chapter 13	Describes how to format output with the PRINT USING statement
Chapter 14	Shows how to use compiler directives
Chapter 15	Explains error handling techniques
Chapter 16	Describes how to use Instruction and Data Space to enable you to run larger tasks
Chapter 17	Describes device-specific input and output on the RSTS/E and RSX operating systems
Chapter 18	Describes how to use memory-resident and object-module libraries
Chapter 19	Describes BASIC–PLUS–2 utilities that you can use to facilitate program development
Chapter 20	Describes techniques you can use to decrease the execution time of your programs
Appendix A	Lists and describes the BASIC–PLUS–2 compile-time error messages
Appendix B	Lists and describes the BASIC–PLUS–2 run-time error messages
Appendix C	Lists the ASCII and RAD-50 character codes and describes data representation formats

Please use the Reader's Comments form in the back of this book to report documentation errors, to comment on how information is presented, or to provide suggestions for future publications.

Conventions

This manual uses lower and uppercase letters, symbols, and mnemonics in syntax diagrams. This symbology aids in providing more concise and exact descriptions of syntactic variables, rules, and formats.

Note

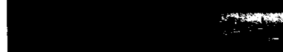
This manual uses DCL as an example of a command-line interpreter. However, MCR is also supported on RSX systems as a command-line interpreter. If you prefer to use MCR as a command-line interpreter, please consult the *RSX-11M/M-PLUS MCR Operations Manual* for the MCR equivalent to DCL commands in this manual.

Convention	Meaning
`Color`	Color in code examples denotes user input.
UPPERCASE letters	Uppercase letters in language syntax denote BASIC–PLUS–2 keywords and must be spelled exactly as shown; you can enter them in either upper or lower case in actual coding.
lowercase letters	Lowercase letters in language syntax denote mnemonics representing user-supplied names or characters.
[ ]	Brackets enclose an optional portion of a format. Brackets around vertically stacked items indicate that you can select one of the enclosed items. You must include all punctuation as it appears in the brackets.
{ }	Braces enclose a mandatory portion of a format. Braces around vertically stacked items indicate that you must choose one of the enclosed items. You must include all punctuation as it appears in the braces.
. . .	A vertical ellipsis indicates that code which would normally be present is not shown.
. . .	An ellipsis indicates that the immediately preceding item can be repeated. An ellipsis following a format unit enclosed in brackets or braces means that you can repeat the entire unit. If repeated items or format units must be separated by commas, the ellipsis is preceded by a comma (, . . .).

The following mnemonics are used in the syntax diagrams:

Mnemonic	Meaning
angle	An angle in radians
array	An array; syntax rules specify whether the bounds or dimensions can be specified
chnl-exp	An I/O channel associated with a file
com	Specific to a COMMON block
cond	Conditional expression; indicates that an expression can be either logical or relational
const	A constant value
data-type	A data type keyword
def	Specific to a DEF function
exp	An expression
file-spec	A file specification
func	Specific to a FUNCTION subprogram
int	An integer value
int-exp	An expression that represents an integer value
int-var	A variable that contains an integer value
label	An alphanumeric statement label
lex	Lexical; used to indicate a component of a compiler directive
line	A statement line; may or may not be numbered
line-num	A statement line number
lit	A literal value, in quotation marks
log-exp	Logical expression
map	Specific to a MAP statement
matrix	A two-dimensional array
name	A name or identifier; indicates the declaration of a name or the name of a BASIC–PLUS–2 structure, such as a SUB subprogram
num	A numeric value
param-list	A parameter list, such as for a SUB subprogram
pass-mech	A valid BASIC–PLUS–2 passing mechanism
real	A floating-point value
rel-exp	Relational expression

Mnemonic	Meaning
str	A character string
str-exp	An expression that represents a character string
str-var	A variable that contains a character string
sub	Specific to a SUB subprogram
target	The target point of a branch statement; either a line number or a label
unsubs-var	Unsubscripted variable, as opposed to an array element
var	A variable



Summary of Technical Changes

The following is a list of the major changes for Version 2.7 of BASIC–PLUS–2:

- The BASIC–PLUS–2 compiler can now run in Instruction and Data (I & D) memory space, thereby improving compilation performance. I & D support is selected when the compiler is built and installed; it does not impact the language syntax.
- The DCL compilation command BASIC has been expanded to allow you to compile multiple BASIC–PLUS–2 programs from DCL.
- A new qualifier, /[NO]BOUNDS, has been added to the environment COMPILE command and to the DCL command BASIC. If you specify /NOBOUNDS, the processing overhead of checking array boundaries on arrays of one or two dimensions is eliminated.
- Command line support has been expanded to allow longer command strings than was previously allowed.
- Several additional performance enhancements have been incorporated to allow faster compilation and run time execution. These enhancements are primarily internal and are not reflected in the language syntax; therefore, they do not require documentation in the manuals.

1 Developing Programs in the BASIC Environment

The BASIC environment has capabilities and features that make the process of program development easier for both novice and expert users. This chapter describes how to work within the BASIC environment.

1.1 Entering the Environment

To enter the BASIC environment, log into the system, enter the DCL command BASIC and press Return. The BASIC command has the following format:

BASIC [/qualifier] . . . [@file-spec [/qualifier] . . .]

@file-spec
The name of an indirect command file containing environment commands. This file specification is optional; it merely gives you a quick way to execute a particular set of environment commands without reentering them each time. The name of the indirect command file must be preceded by an at sign (@). If you do not specify a file type, BASIC–PLUS–2 assumes the file has a file type of CMD. See Section 1.4 for more information about specifying indirect command files with the BASIC command.

Note that there is another version of the BASIC command that compiles one or more BASIC–PLUS–2 source files (.B2S) at the DCL command level. In the BASIC compilation command, the (@) does not precede the file specification and all activity occurs at DCL command level; the BASIC environment is not entered.

/qualifier
The name of a qualifier that sets an environment default. The default remains in effect until you either use the SET command to override the default or exit from the environment.

Command Qualifier	Default
/[NO]BOUND	/BOUND
/[NO]CHAIN	See text.
/[NO]CROSS_REFERENCE [= [NO]KEYWORDS]	/NOCROSS_REFERENCE
/[NO]DEBUG	/NODEBUG
/[NO]FLAG[= [NO]DECLINING]	/FLAG=DECLINING
/[NO]LINES	/LINES
/[NO]LIST	/NOLIST
/[NO]MACRO	/NOMACRO
/[NO]OBJECT	/OBJECT
/SCALE = int-const	/SCALE=0
/VARIANT = int-const	/VARIANT=0
/[NO]WARNINGS	/WARNINGS

See Chapter 2 for a description of each of these qualifiers.

When you specify the BASIC command, BASIC–PLUS–2 responds with an identification line and the BASIC2 prompt. For example:

```
$ BASIC
PDP-11 BASIC-PLUS-2 V2.7-00

BASIC2
```

Note

BP2 is the default name for the BASIC–PLUS–2 compiler. However, your system manager may have chosen another name for the compiler while installing BASIC–PLUS–2. If you have trouble entering the BASIC environment while using the BASIC command, see your system manager for help.

Once you are in the BASIC environment, you interact directly with the compiler. In this mode of operation, you can enter any of the following:

- BASIC–PLUS–2 program lines
- Immediate mode statements
- Compiler commands and qualifiers

When you enter program statements, BASIC–PLUS–2 stores them in ascending line number sequence as part of the current program in memory. If you enter a program line with the same line number as an existing program line, the new line replaces the old one.

When you create a program in the environment, the first line of the program must have a line number. If you enter a subsequent program line without a line number, you must precede it with a space or a tab. Inside the environment, only those program lines that begin with line numbers can start in the first character position on a line.

If a program line is too long for one text line, you can continue it by entering an ampersand (&) and pressing Return. (Note that only spaces and tabs are valid between the ampersand and the carriage return.)

See Section 1.7 for more information about immediate mode statements and Section 1.8 for more information about commands you can use in the BASIC–PLUS–2 environment.

1.2 Creating Programs

There are two ways to create a program you can use in the environment. You can create the program interactively while inside the environment or you can create the program using a text editor from DCL level.

1.2.1 Creating a Program Interactively

To create a new program, enter the BASIC environment and specify the NEW command with the file specification of the program you want to create. For example:

Example

```
$ BASIC

PDP-11 BASIC-PLUS-2 V2.7-00

BASIC2

NEW FIRSTTRY
```

In this example, the NEW command creates the program FIRSTRY.B2S.

If you do not specify a file specification with the NEW command, BASIC–PLUS–2 assigns the name NONAME.B2S to the program. If you supply a file name but do not specify a file type, BASIC–PLUS–2 assigns a file type of B2S, by default.

Once you enter the NEW command, you can enter in your program. When you finish the program, you can keep it for future use with the SAVE command. The SAVE command writes the program to your current default directory.

The following environment session creates the program FIRSTTRY.B2S and then saves it for future use:

Example

```
$ BASIC

PDP-11 BASIC-PLUS-2 V2.7-00

BASIC2

NEW FIRSTTRY

BASIC2

10  PRINT "Please enter three numbers"
        INPUT A, B, C
        PRINT "Their average is"; ( A + B + C ) / 3
        END
SAVE
```

When you create a program in the environment, that program remains in memory until you either use the OLD command to bring another program into memory or until you exit from the environment. You can display the program currently in memory by using the LIST command.

If you make an error in your program, you can either re-enter the program line containing the error, or you can edit the program line by using the EDIT command. If you specify the EDIT command with no parameters, BASIC–PLUS–2 places you in editing mode where you can enter editing mode commands. See the *BASIC–PLUS–2 Reference Manual* for a description of the EDIT command as well as the BASIC–PLUS–2 editing mode commands.

1.2.2 Creating a Program Using a Text Editor

Instead of entering your program directly in the environment, you have the option of creating your program with a text editor accessed from DCL. Once you create the program, you can either enter the BASIC environment and use the OLD command to read your program into memory, or compile your program at DCL level. Chapter 2 discusses how to compile your programs at DCL level.

1.3 Running Programs

Once you create a program, you can enter the RUN or RUNNH command to compile, link, and execute your program. (RUNNH suppresses header information such as the name of the program and the time of day.) For example:

```
BASIC2

OLD FIRSTTRY

BASIC2

RUN
```

The RUN command compiles, links, and executes the program you create.

1.4 Indirect Command Files

Instead of entering the BASIC environment and entering commands interactively, you have the option of placing your environment commands in an indirect command file. An indirect command saves you time because you can use it over and over again to execute the same sequence of commands without having to enter them in.

To create an indirect command file, you use a text editor and enter each command you want to execute on a separate line. Then, you exit from the indirect command file and execute it from DCL level with the BASIC command. For example:

```
$BASIC @MYIND
```

You must precede the file specification of an indirect command with an at sign (@). If you do not specify a file type with the indirect command file, BASIC-PLUS-2 assumes a file type of CMD by default.

The following indirect command file, for example, creates the program AVER.B2S, saves it, and then executes it:

```
NEW AVER
10  PRINT "Please enter three numbers"
        INPUT A, B, C
        PRINT "Their average is"; ( A + B + C ) / 3
        END
SAVE
RUN
```

When you execute an indirect command file, you remain at DCL level. You cannot invoke an indirect command file from within another indirect command file.

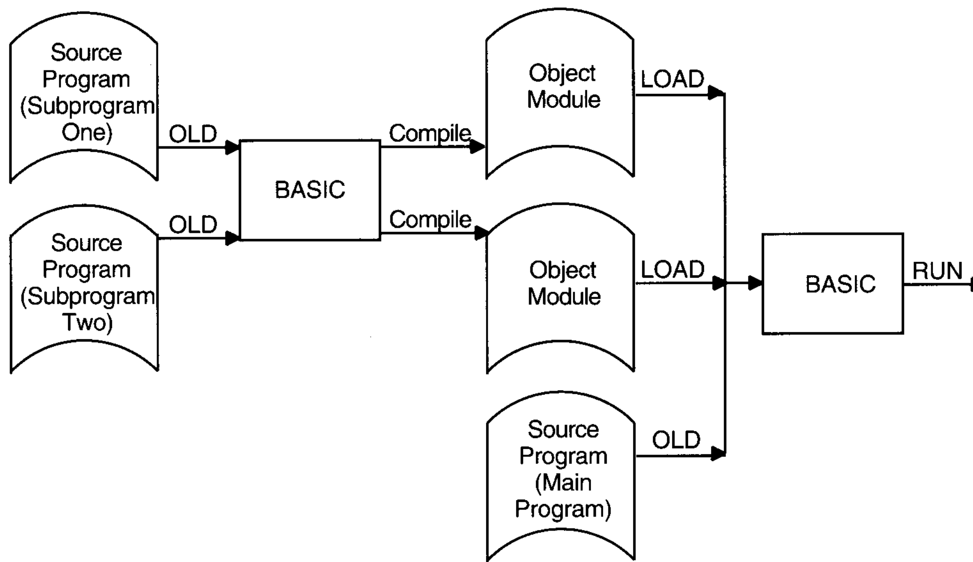
1.5 Multiple-Unit Programs

You can execute multiple-unit programs while in the BASIC environment. To execute multiple-unit programs, follow these steps:

1. Compile all subprograms to generate object modules
2. Use the OLD command to read the main program into memory
3. Use the LOAD command to read the subprogram object modules into memory
4. Enter the RUN command

Figure 1–1 illustrates how to execute multiple-unit programs.

Figure 1–1 Running Multiple-Unit Programs



The following is an example of a program that contains multiple units:

Example

Main Program

```
10       REM This program calls SUBPROGRAM SB1
20       PRINT "NOW IN MAIN PROGRAM"
30       CALL SB1
40       PRINT "BACK IN MAIN PROGRAM"
50       END
```

Subprogram

```
10       SUB SB1
20       PRINT "NOW IN SUBPROGRAM"
30       SUBEND
```

To execute these programs in the BASIC environment, enter the following commands:

```
OLD SB1
BASIC2

COMPILE
BASIC2

OLD MAIN
BASIC2

LOAD SB1
BASIC2

RUN
```

Output

```
NOW IN MAIN PROGRAM
NOW IN SUBPROGRAM
BACK IN MAIN PROGRAM
BASIC2
```

1.6 Exiting from the Environment

When you exit from the BASIC–PLUS–2 environment, the program you are currently working on is erased. Therefore, before you exit you should make certain to save your program if you want to use it again. If you are entering a new program, you can use the BASIC–PLUS–2 SAVE command. If you are revising an old program, you should use the BASIC–PLUS–2 REPLACE command.

To exit from the BASIC–PLUS–2 environment, specify the EXIT command. For example:

```
EXIT
```

Once you exit from the BASIC–PLUS–2 environment, the operating system displays the DCL dollar sign ($) prompt where you can enter DCL commands.

1.7 Immediate Mode

You do not have to write a complete program in order to use BASIC–PLUS–2. Many statements are executable in *immediate mode.*

Immediate mode statements are BASIC statements that are executed immediately after you press the Return key. Immediate mode statements cannot be preceded by a line number, space, or tab and can be used only if you are working directly in the environment.

In the following example, BASIC–PLUS–2 interprets the first line as a part of a larger program because it begins with a line number. This line will not execute until a RUN command is specified. The second line does not begin with a line number, a space, or a tab. Therefore, BASIC–PLUS–2 treats the line as an immediate mode statement and immediately displays the specified text.

Example

```
10  PRINT 'This is an executable BASIC--PLUS--2 statement'
PRINT 'This is an immediate mode statement'
```

Output

```
This is an immediate mode statement

BASIC2
```

The BASIC2 prompt indicates that BASIC–PLUS–2 is ready to receive compiler commands, immediate mode statements, or new program lines.

You can precede each executable statement with a backslash (\ ). You can also have more than one BASIC–PLUS–2 statement on a line if you separate them with a backslash; however, programs with backslashes are often difficult to read as seen in the following example:

Example

```
BASIC2

A = (54.37 / 1.25 ) \ B = ( 328.15^2) \ PRINT ( B / A )
 2475.69
```

BASIC–PLUS–2 compiles and executes each immediate mode statement as if it were a self-contained program. For example:

Example

```
BASIC2
PRINT PI * 67.3
 211.421
```

Each immediate mode line exists by itself, and any variables used by the statements on that line are temporary. For example:

Example

```
BASIC2
A = 2^5 \ PRINT A
 32

BASIC2

PRINT A
 0
```

In this example, the second PRINT statement causes BASIC–PLUS–2 to display a zero because the compiler treats *A* as a new variable, and initializes it to zero.

You can use the IF, WHILE, UNTIL, UNLESS, and FOR statement modifiers in immediate mode statements. The following example, for instance, uses the FOR statement modifier to generate a table of square roots:

Example

```
BASIC2

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
BASIC2
```

Certain statements are invalid in immediate mode. In general, invalid statements are statements that require the allocation of new storage, or statements that make no sense in the context of a single line. If you try to execute such a statement, BASIC–PLUS–2 signals the error "Illegal in immediate mode".

1.8 Environment Commands

Compiling is the process of translating a source program to an object module. An object module is an intermediate step between source code and an executable image. It contains information that the linker uses to create an image.

You can compile, link, and execute your programs in the environment by entering the RUN command. This greatly reduces the number of steps you have to go through to develop BASIC–PLUS–2 programs. When you are satisfied with a portion of code, you can simply run that program to examine whether or not it functions as expected. If you use the COMPILE command instead, you can eliminate all compile-time errors before you link and execute the program.

BASIC–PLUS–2 has certain defaults that are in effect each time you enter the BASIC environment. Unless you explicitly override these defaults, they remain in effect until you leave the environment. You can see a listing of these defaults by entering the SHOW command when in the environment. The following example displays the standard BASIC environment defaults that are in effect when you enter the environment:

```
SHOW
PDP-11 BASIC-PLUS-2 V2.7-00 using EIS with run support

ENVIRONMENT INFORMATION:            RMS FILE ORGANIZATION:
    Current edit line : 0              NO Index
    NO Modules loaded                  NO Relative
    NO Main module loaded              NO Sequential
                                       NO Virtual

DEFAULT DATA TYPE INFORMATION:      LISTING FILE INFORMATION:
    Data type : REAL                   NO Source
    Real size : SINGLE                 NO Cross Reference
    Integer size : WORD                NO Keywords
    Scale factor : 0                      60 lines by 132 columns

COMPILATION QUALIFIERS:             BUILD QUALIFIERS:
       Object                          NO Dump
    NO Macro                           NO Map
       Lines                           NO Cluster
       Warnings                        NO I- and D-Space
    NO Debug records                   Task extend  : 512
    NO Syntax checking                 RMS ODL file : LB:[1,1]RMS11X
    Flag : Declining                   BP2 Disk lib : LB:[1,1]v27OTS
    Variant : 0                        RMS Resident lib : NONE
                                       BP2 Resident lib : NONE

BASIC2
```

You can override any of these defaults with qualifiers to the COMPILE, BUILD, or SET commands, or with the OPTION statement in your program. You can also change the environment defaults you receive by creating an initialization file. See Section 1.9 for more information.

Table 1–1 lists the BASIC–PLUS–2 environment commands and defines their function. The sections following the table describe each environment command and provide an example of its use.

Table 1–1 Environment Commands

Command	Description
$ system-command	Starts a subprocess to execute the specified DCL command.
APPEND	Merges the specified program with the program currently in memory.
BRLRES	Changes the default memory-resident library.
BUILD	Generates an overlay description file and a command file.
COMPILE	Generates an object module (file type OBJ) from a BASIC–PLUS–2 source program.
DELETE	Erases the specified line or lines from a BASIC–PLUS–2 source program.
DSKLIB	Changes the default BASIC–PLUS–2 disk-resident library.
EDIT	Changes source text or calls a text editor.
EXIT	Returns to DCL command level.
EXTRACT	Extracts the specified program line or lines from a BASIC–PLUS–2 source program and deletes the rest.
HELP	Provides online help.
IDENTIFY	Causes BASIC–PLUS–2 to print an identification header on your screen.
INQUIRE	Identical to the HELP command.
LIBRARY	Changes the default BASIC–PLUS–2 memory-resident library.
LIST	Displays the current source program on your screen.
LISTNH	Displays the current source program without header information.
LOAD	Loads an object module into memory.
LOCK	Specifies default values for environment command qualifiers (identical to the SET command).

(continued on next page)

Table 1–1 (Cont.) Environment Commands

Command	Description
NEW	Clears memory for the creation of a new program and assigns a new program name.
ODLRMS	Changes the default RMS overlay description file (ODL).
OLD	Reads a specified BASIC–PLUS–2 source program into memory.
RENAME	Changes the name of the program currently in memory.
REPLACE	Replaces a stored program with the program currently in memory.
RMSRES	Changes the default RMS memory-resident library.
RUN	Executes the program currently in memory, or a specified BASIC–PLUS–2 source program. The program in memory can be: – A BASIC–PLUS–2 source program placed in memory with the OLD command – One or more object modules placed in memory with the LOAD command – A combination of the first two
RUNNH	Identical to RUN but does not display header information.
SAVE	Creates a copy of the current source program on a specified device.
SCALE	Controls accumulated round-off errors for numeric operations.
SCRATCH	Erases the current program and any loaded object modules.
SEQUENCE	Generates line numbers for input text.
SET	Specifies default values for environment command qualifiers.
SHOW	Displays the current default environment qualifiers.
UNSAVE	Deletes a specified file.

See the *BASIC–PLUS–2 Reference Manual* for a complete description of all BASIC–PLUS–2 environment commands.

1.8.1 The $ System-Command

You can enter a DCL command while in the environment by preceding it with a dollar sign ($). BASIC–PLUS–2 passes the command to the system for execution. On RSX systems, the program currently in memory does not change. When you enter a system-command on RSTS/E systems, however, the program currently in memory is deleted. Therefore, you should specify the SAVE command before entering a system-command on RSTS/E systems.

1.8.2 The APPEND Command

The APPEND command merges a BASIC–PLUS–2 source program with the program currently in memory. The program in memory must be a BASIC–PLUS–2 source program that has been placed in memory with the OLD command and entered in the environment. The program must also contain at least one line number.

If both programs contain a line with the same number, the appended program line replaces the current program line.

If you enter APPEND without specifying a file name, BASIC–PLUS–2 prompts with:

```
Append file name--
```

You should respond with a file name. If you respond by entering the Return key, BASIC–PLUS–2 searches for a file called NONAME with the default file type of B2S. If the compiler cannot find the file, it signals an error.

The APPEND command does not change the name of the program in memory.

1.8.3 The BRLRES Command

The BRLRES command allows you to specify a memory-resident library to be used when you link a program. When you use the BUILD command, BASIC–PLUS–2 includes the specified library in the Task Builder command file. Your system manager chooses the default library for the BRLRES command during installation.

If you enter BRLRES without specifying an argument, BASIC–PLUS–2 prompts with:

```
File spec [NONE]--
```

You can either specify a memory-resident library, or press Return to take the default, NONE. NONE specifies that no memory-resident library is to be used when linking the program.

1.8.4 The BUILD Command

The BUILD command generates a command (CMD) file and an overlay description language (ODL) file for the Task Builder. The CMD file contains instructions that enable the Task Builder to link your program module or modules with libraries and other routines. The ODL file specifies how segments of the linked program are overlaid when you run it. For example, the following BUILD command creates the files MAIN.CMD and MAIN.ODL. The Task Builder uses these files to control the creation of an executable task image.

```
BUILD MAIN,SUB1,SUB2
```

The BUILD command has several command qualifiers. For example, the /IDS qualifier causes the Task Builder to build the task in I- and D-Space and the /RELATIVE qualifier causes the Task Builder to include the code needed for relative file operations. For a complete description of BUILD command qualifiers, see the *BASIC–PLUS–2 Reference Manual.*

1.8.5 The COMPILE Command

When you compile a program in the BASIC environment, there are three levels at which you can specify options for the compiler:

- You can accept the defaults of the BASIC environment as options
- You can specify options with qualifiers to the COMPILE or SET command
- You can specify options in the source program with the OPTION statement

The COMPILE command creates an object module from a source program in memory. You can control the compilation of your program with the COMPILE command and its qualifiers. These qualifiers duplicate many of the qualifiers available to the DCL command BASIC. You can abbreviate all COMPILE qualifiers to four letters. For example, you can compile a program currently in memory and specify the creation of a listing file:

```
COMPILE/LIST
```

The following two commands both specify that a listing file should be created. Note that the SET command sets a particular default until you leave the BASIC environment or until you specify a different default for that value, whereas the qualifiers to the COMPILE command set the defaults only for that particular compilation.

```
SET/LIST
COMPILE/LIST
```

If you do not specify any qualifiers with the SET command, BASIC–PLUS–2 resets the defaults to the values that were in effect when you entered the BASIC environment. If you do specify qualifiers with the COMPILE command, the BASIC environment default values remain the same, but your program is compiled using the specified defaults.

See the *BASIC–PLUS–2 Reference Manual* for a complete list and description of the COMPILE command qualifiers.

1.8.6 The DELETE Command

The DELETE command removes a specified line or lines from the source program currently in memory. If you separate line numbers with commas, BASIC–PLUS–2 deletes only the specified program lines. If you separate line numbers with a hyphen (-), BASIC–PLUS–2 deletes the specified program lines and all program lines between them. For example:

DELETE 10	Removes line 10 from the program
DELETE 50, 100	Removes lines 50 and 100 from the program
DELETE 50, 100-190	Removes line 50 and lines 100 through 190 from the program

If you do not specify a line number, the DELETE command is ignored.

1.8.7 The DSKLIB Command

The DSKLIB command lets you select a disk-resident object module library to be used when you build your program. When you use the BUILD command, BASIC–PLUS–2 includes the specified library in the Task Builder command file.

If you do not supply a file specification or library name with the DSKLIB command, BASIC–PLUS–2 prompts with:

```
File spec [default-lib]--
```

If you press Return in response to this prompt, BASIC–PLUS–2 uses the default disk-resident library.

1.8.8 The EDIT Command

The EDIT command replaces text in the current program with text you supply in the command. When you supply text as an argument to the EDIT command, you are editing in *line mode*. If you enter EDIT with no argument, BASIC–PLUS–2 enters *editing mode* where you can enter editing mode commands. See the *BASIC–PLUS–2 Reference Manual* for a complete description of all editing mode commands.

The following are examples of editing in line mode:

EDIT 100 /LEFT$/RIGHT$/	Replaces the first occurrence of LEFT$ with RIGHT$ on line 100.
EDIT	Invokes the default editor and reads the current program into the editor's buffer.

EDIT 2000	Lists line 2000 (line 2000 becomes the default EDIT line).
EDIT 30 /LEFT$/RIGHT$/,3	Starts the search on the third text line of program line 30 and replaces the first occurrence of LEFT$ with RIGHT$.
EDIT 300/LEFT$//2	Removes the second occurrence of the string LEFT$ from line 300. Note that you must specify delimiters around the null replacement string. Otherwise, the EDIT command would replace the first occurrence of LEFT$ with 2.

The following is an example of editing in editing mode:

EDIT SUBSTITUTE /REM/!/ EXIT	Substitutes an exclamation mark (!) comment field for a REM statement and exits from editing mode.

1.8.9 The EXIT Command

The EXIT command clears memory and returns control to DCL command level. If you modify a program and issue the EXIT command before you copy it to disk with the SAVE or REPLACE command, BASIC–PLUS–2 signals "Unsaved change has been made, Ctrl/Z or EXIT to exit." This message warns you that any changes will be lost if you do not save the program. You can then store the program or reenter the EXIT command (or press Ctrl/Z) to exit from BASIC–PLUS–2.

1.8.10 The EXTRACT Command

The EXTRACT command removes a specified line or lines from the source program currently in memory and deletes the rest. If you separate line numbers with commas, BASIC–PLUS–2 extracts only the specified program lines. If you separate line numbers with a hyphen (-), BASIC–PLUS–2 extracts the specified program lines and all program lines between them. For example:

EXTRACT 10	Extracts line 10 from the program
EXTRACT 50, 100	Extracts lines 50 and 100 from the program
EXTRACT 50, 100-190	Extracts line 50 and lines 100 through 190 from the program

If you do not specify a line number, the EXTRACT command is ignored.

1.8.11 The HELP Command

The HELP command lets you display the contents of the BASIC–PLUS–2 HELP library on the terminal. Entering HELP causes the HELP facility to display a long list of BASIC–PLUS–2 commands and language topics for which there is help available. You are then prompted to name a command or topic with the following prompt:

```
Topic?
```

To obtain help on the environment commands, you can enter COMMANDS at the "Topic?" prompt. A list of commands is displayed on your terminal followed by the prompt "COMMANDS Subtopic?." When you enter a command name in response to this prompt, the HELP facility displays the following:

- An explanation of the command's purpose
- An example of its use
- A list of any further subtopics available

You can also display help text for BASIC–PLUS–2 errors. Help for BASIC–PLUS–2 errors is grouped in three categories: run-time errors, debugger errors, and error handling.

1.8.12 The IDENTIFY Command

The IDENTIFY command prints a header containing the BASIC–PLUS–2 compiler name and version number. For example:

```
IDENTIFY

PDP-11 BASIC-PLUS-2 V2.7-00

BASIC2
```

1.8.13 The INQUIRE Command

The INQUIRE command is identical to the HELP command. See the HELP command for more information.

1.8.14 The LIBRARY Command

The LIBRARY command allows you to specify a memory-resident library to be used when you link your program. When you use the BUILD command, BASIC–PLUS–2 includes the specified library in the Task Builder command file. Your system manager chooses the default library for the LIBRARY command during installation.

If you do not supply an argument to the library command, BASIC–PLUS–2 prompts with:

```
File spec [default-lib]--
```

You can either supply a library name or the keyword NONE in response to this prompt. NONE specifies that no library is to be used when linking the program. If you press Return in response to the prompt, BASIC–PLUS–2 uses the default memory-resident library.

1.8.15 The LIST and LISTNH Commands

The LIST and LISTNH commands display a specified line or lines. If you enter LIST or LISTNH without specifying line numbers, BASIC–PLUS–2 displays a copy of the source program currently in memory, in ascending line number order.

The LIST command prints a header displaying the program name and the current time and date before displaying the specified lines. The LISTNH command suppresses the header information and prints the specified lines only. For example:

LIST 10	Displays header information, then displays line 10.
LISTNH 50, 100	Displays lines 50 and 100.
LIST 50, 90, 100-190	Displays header information, then displays lines 50, 90, and 100 through 190.

1.8.16 The LOAD Command

The LOAD command makes an object module available for execution with the RUN command. You can load only object files created by BASIC–PLUS–2.

The LOAD command accepts multiple device, directory, and file specifications. The LOAD command deletes all previously loaded object files; therefore, to load several files at the same time, you must separate the file specifications with plus signs. Multiple file specifications separated with commas cause each file to be loaded separately, thereby deleting the previously loaded file.

```
LOAD OLD1 + OLD2 + OLD3
BASIC2
RUN
```

The above example loads the files OLD1.OBJ, OLD2.OBJ, and OLD3.OBJ for execution. These object files are not linked with the current program or executed until you issue the RUN command. Therefore, run-time errors in the loaded modules are not detected until you execute the program.

Each device and directory specification applies to all following file specifications until you specify a new directory or device. For example:

```
LOAD DU1:[SMITH]PROG3+[JONES]PROG4+DU2:PROG5
```

This command loads three object files:

- PROG3 from the directory SMITH on the device DU1:
- PROG4 from the directory JONES on DU1:
- PROG5 from the directory JONES on DU2:

1.8.17 The LOCK Command

The LOCK command changes default values for COMPILE command qualifiers. It is equivalent to the SET command. The following command specifies that all subsequent compilations use double-precision floating-point numbers as the default. You can use any valid COMPILE command qualifier as an argument to LOCK.

```
LOCK /DOUBLE
```

```
BASIC2
```

1.8.18 The NEW Command

The NEW command clears the memory and assigns a name to a program to be entered. The following command assigns the name PROG1 to the program. You can then enter program lines.

```
NEW PROG1
```

If you do not specify a name, BASIC–PLUS–2 issues the following prompt:

```
New file name--
```

You should respond with a name. If you press the Return key in response to the prompt, BASIC–PLUS–2 assigns the name NONAME.

1.8.19 The ODLRMS Command

The ODLRMS command allows you to select an overlay description (ODL) file to describe the RMS overlay structure to be used when you link your program. When you use the BUILD command, BASIC–PLUS–2 includes the specified ODL file in the Task Builder command file. Your system manager chooses the default ODL file during installation.

1.8.20 The OLD Command

The OLD command brings a previously created BASIC–PLUS–2 source file into memory. The following command reads PROG1.B2S into memory.

```
OLD PROG1
```

If you do not specify a file name, BASIC–PLUS–2 issues the prompt:

```
Old file name--
```

You should respond with a file name. If you do not specify a file type, BASIC–PLUS–2 reads a file with the specified file name and the default file type. If you press the Return key in response to the prompt, BASIC–PLUS–2 searches for a file with the default file name and default file type: NONAME.B2S.

1.8.21 The RENAME Command

The RENAME command assigns a new name to the program currently in memory. For example, the following command sequence brings a program named PROG1 into memory and changes its name and directory:

```
OLD [OKAY]PROG1

BASIC2

RENAME [KAY]PROG2
```

The name of the program is changed to PROG2. The disk file from which the file was read remains unchanged.

1.8.22 The REPLACE Command

The REPLACE command writes the program in memory to a specified device. If you do not specify a device, the program is written to the default disk with the file name of the program currently in memory. If a file of the same name already exists, BASIC–PLUS–2 supersedes the old version. If you specify a file name with the REPLACE command, the current program is stored on disk under the file name you specify.

1.8.23 The RUN and RUNNH Commands

The RUN command executes a program. This program can be any one of the following:

- The current program
- One or more object modules placed in memory with the LOAD command

- A combination of the first two
- A specified BASIC–PLUS–2 source program

If you do not supply an alternative file specification, BASIC–PLUS–2 executes the program in memory.

```
BASIC2

OLD
Old file name--PROG1
BASIC2

RUN
```

The RUN command compiles, links, and executes PROG1. It prints a header displaying the program name and the current date and time. To execute a program without displaying this header, enter RUNNH.

The RUN command does not create an object module file or a list file. It uses whatever qualifiers have been set. The following qualifiers are always in effect for the RUN and RUNNH commands. For a complete list of RUN command qualifiers, see the *BASIC–PLUS–2 Reference Manual.*

- /NOCROSS
- /NODEBUG
- /NOLIST
- /NOMACHINE
- /NOOBJECT

The RUN command can invoke only BASIC–PLUS–2 procedures.

1.8.24 The SAVE Command

The SAVE command copies a BASIC–PLUS–2 source program from memory to a file. You can specify a storage device, a file name, and a file type in the SAVE file-spec. For example, if you enter the following program, a SAVE command causes BASIC–PLUS–2 to arrange the program in ascending line number order and copy it to a file on DU1: in the current default directory with a file name of TEST and the default file type B2S:

Example

```
30 PRINT "THIS IS A TEST"
10 REM THIS IS A TEST
SAVE DU1:TEST.B2S
```

BASIC–PLUS–2 saves the program on DU1: in the current default directory with a file name of TEST and a file type of B2S. If the program in memory has no name and you issue the SAVE command with no argument, BASIC–PLUS–2 copies the program to a file named NONAME with the default file type in your current default device and directory. Note that if you perform a RENAME operation, before you issue the SAVE command followed by no argument, BASIC–PLUS–2 still copies the program to the current default directory.

1.8.25 The SCALE Command

The SCALE command can overcome accumulated round-off errors by multiplying double-precision floating-point values by 10 raised to the specified scale factor before storing them. In the following example, the value of scaled arithmetic is two. When a program is compiled, all double-precision, floating-point numbers will either be multiplied by 100 or divided by 100, where required:

```
SCALE 2
Scale factor has been set to 2

BASIC2
```

1.8.26 The SCRATCH Command

The SCRATCH command clears memory by doing one of the following:

- Resetting the program name to NONAME
- Removing any object files previously loaded with the LOAD command
- Removing the source file currently in memory

1.8.27 The SEQUENCE Command

The SEQUENCE command automatically generates line numbers for input text. After a SEQUENCE command, BASIC–PLUS–2 prompts with a line number and prompts again after each source line you enter. If you press Ctrl/Z (either in response to the line number prompt or at the end of a program line), BASIC–PLUS–2 stops prompting and you can enter source text in the normal way. If you specify a starting line number that already contains a statement, BASIC–PLUS–2 signals “Attempt to sequence over existing statement” and returns to normal input mode.

1.8.28 The SET Command

The SET command specifies defaults for compiler command qualifiers. For example:

```
SET /SINGLE

BASIC2
```

This command makes /SINGLE the default for the COMPILE or RUN command, thereby making SINGLE the default data type for all untyped values. Entering SET with no qualifier resets the defaults to their state when you entered into the BASIC environment.

For a full list of SET command qualifiers, see the *BASIC–PLUS–2 Reference Manual*.

1.8.29 The SHOW Command

The SHOW command displays the current default qualifiers and user libraries.

```
SHOW
PDP-11 BASIC-PLUS-2 V2.7-00 using EIS with run support

ENVIRONMENT INFORMATION:              RMS FILE ORGANIZATION:
    Current edit line : 0                NO Index
    NO Modules loaded                    NO Relative
    NO Main module loaded                NO Sequential
                                         NO Virtual

DEFAULT DATA TYPE INFORMATION:        LISTING FILE INFORMATION:
    Data type : REAL                     NO Source
    Real size : SINGLE                   NO Cross Reference
    Integer size : WORD                  NO Keywords
    Scale factor : 0                        60 lines by 132 columns

COMPILATION QUALIFIERS:               BUILD QUALIFIERS:
       Object                            NO Dump
    NO Macro                             NO Map
       Lines                             NO Cluster
       Warnings                          NO I- and D-Space
    NO Debug records                     Task extend  : 512
    NO Syntax checking                   RMS ODL file : LB:[1,1]RMS11X
    Flag : Declining                     BP2 Disk lib : LB:[1,1]v27OTS
    Variant : 0                          RMS Resident lib : NONE
                                         BP2 Resident lib : NONE
BASIC2
```

The ENVIRONMENT INFORMATION section gives you the following information:

- No edit line is currently being edited.
- No object modules are currently loaded in the environment.

- No main program module is currently loaded in the environment.

The DEFAULT DATA TYPE INFORMATION display gives you the following information:

- The default data type is REAL.
- The default size for floating-point numbers is SINGLE and the default size for integers is WORD.
- There is no scale factor in effect.

The COMPILATION QUALIFIERS section gives you the following information:

- An object file is produced.
- No macro source code file is generated.
- Line number information is included in the object file.
- Warning or informational error messages are displayed.
- No debug records are included in the object module. This means you cannot access program symbols with the BASIC–PLUS–2 debugger.
- Line-by-line syntax checking is disabled.
- Declining features are reported.
- The VARIANT value is zero.

The RMS FILE ORGANIZATION display gives you the following information:

- No support is provided for RMS indexed file operations.
- No support is provided for RMS relative file operations.
- No support is provided for RMS sequential file operations.
- No support is provided for RMS virtual file operations.

The LISTING FILE INFORMATION display tells you which parts of the program listing are included if you create a compilation listing:

- No source program is listed.
- No cross-reference information is listed.
- No qualifiers are in effect when the program is compiled.
- The page size of the listing file is 60 lines in length and 132 columns in width.

The BUILD QUALIFIERS section gives you the following information:

- No dump file is generated if your program aborts.
- No allocation map file is generated.
- The Task Builder will not cluster the default BASIC–PLUS–2 and RMS–11 resident libraries.
- The Task Builder will not use I- and D-Space.
- The Task Builder will use the RMS ODL file LB:[1,1]RMS11X to overlay segments of the RMS–11 object module library.
- The Task Builder will use the disk-resident object-module library, LB:[1,1]v27OTS, to link your program.
- No default BASIC–PLUS–2 memory-resident library is in effect.
- No default RMS–11 memory-resident library is in effect.

You can change these defaults with qualifiers to the COMPILE, SET, or BUILD commands. See the *BASIC–PLUS–2 Reference Manual* for a complete description of qualifiers to these commands. You can also change the environment defaults you receive by creating an initialization file. See Section 1.9 for information on creating an initialization file.

1.8.30 The UNSAVE Command

The UNSAVE command deletes the specified version of a file from disk. If you do not specify a file, UNSAVE deletes the disk file associated with the program currently in memory. If you do not specify a version number, UNSAVE deletes the newest version. For example:

```
OLD PROG1

BASIC2

UNSAVE

BASIC2
```

The OLD command copies a program named PROG1.B2S from disk to memory. The UNSAVE command deletes the program from disk.

You can delete a BASIC–PLUS–2 source program other than the one in memory by specifying the program name. The following command deletes the most recent version of the file PROG2.B2S.

```
UNSAVE PROG2
```

To delete a file other than a source program, specify the file name and file type. The following command deletes the newest version of the object module generated from the compilation of PROG2.

```
UNSAVE PROG2.OBJ
```

1.9 Setting Environment Defaults with an Initialization File

Using environment commands and qualifiers is useful when the new defaults affect only a few compilations. However, if you use a particular set of defaults frequently, changing the defaults each time you enter the BASIC–PLUS–2 environment can be tedious. You can have certain defaults take effect automatically whenever you enter the BASIC–PLUS–2 environment by creating an *initialization file.*

Whenever you invoke BASIC–PLUS–2, it searches for two initialization files named BP2INI.BP2. BASIC–PLUS–2 searches for the first file in the system account, LB:, and then searches in your current default directory for the second file.

The system manager sets defaults for all users on the system by creating an initialization file in the system account. These defaults are displayed when you use the SHOW command. Each time the BASIC–PLUS–2 compiler is invoked, it executes the commands in the system BP2INI.BP2 file.

You can override the defaults set in the system initialization file by creating your own initialization file in your default account. Like the system initialization file, it must be named BP2INI.???, where ??? is the name used to invoke BASIC–PLUS–2, as defined at installation time. The default compiler name is BP2.

To create an initialization file, you use a text editor. The file can contain any of a subset of BASIC–PLUS–2 environment commands. Each command must appear on a separate line. Comments are not allowed; however, you can use blank lines to separate the commands for readability.

The following is an example of a user initialization file:

```
SET /LIST
SET /DOUBLE/LONG
SCALE 4
SHOW
```

When this initialization file is executed, the compiler uses the following defaults:

1. The COMPILE command produces a source listing.
2. The default size for all floating-point data is DOUBLE. The default size for all integers is LONG.
3. The scale factor is 4.
4. The compiler displays a list of the new default environment settings.

You cannot tell that an initialization file is executing unless an error occurs in the file. Therefore, you should include the SHOW command to display the present defaults. The SHOW command should be the last command in the file so that the most recently set defaults are displayed.

Table 1–2 lists the BASIC–PLUS–2 environment commands that you can use in an initialization file.

Table 1–2 Commands Allowed in an Initialization File

Command	Function
BRLRES	Specifies the default BASIC–PLUS–2 memory-resident library
DSKLIB	Specifies the default BASIC–PLUS–2 object module library
LIBRARY	Specifies the default BASIC–PLUS–2 memory-resident library
LOCK	Sets qualifiers to the COMPILE or BUILD command
ODLRMS	Specifies the default RMS ODL file
RMSRES	Specifies the default RMS memory-resident library
SCALE	Sets the scale factor
SET	Sets qualifiers to the COMPILE or BUILD command
SHOW	Displays the current defaults

Note that if you use environment commands that specify a default library (such as BRLRES) in an initialization file, you must remember to supply a file specification for the library. If you do not specify a library file specification with the command, BASIC–PLUS–2 signals the error "No file specified for command in initialization file."

2

Developing Programs at DCL Command Level

The process of developing a BASIC–PLUS–2 program involves four steps: creating, compiling, linking, and running. You can accomplish each of these steps using DCL commands. This chapter describes how to create, compile, link, and run a BASIC–PLUS–2 program.

2.1 Using EDT to Create a BASIC–PLUS–2 Program

PDP–11 EDT is an interactive general-purpose text editor that offers three editing modes: keypad, nokeypad, and line. Both keypad and nokeypad modes are screen editors. Keypad mode uses the numeric keypad that appears to the right of your main keyboard. With nokeypad mode, you enter commands on a command line, which EDT processes when you press Return. Line mode focuses on the line as the basic unit of text. The appearance of a line mode asterisk prompt (*) indicates that you can enter a line mode command. When you begin your editing session, editing in line mode is the default. Unlike line mode, keypad mode and nokeypad mode continuously display the contents of the file on your screen.

The following command line invokes the EDT editor and creates the file, PROG1.B2S.

```
$ EDIT/EDT PROG1.B2S
```

To change from line mode to keypad mode, enter the CHANGE command at the asterisk prompt. To return to line mode from keypad mode, press Ctrl/Z. To change from line mode to nokeypad mode, enter the SET NOKEYPAD command and then enter the CHANGE command.

If you are in the middle of an editing session and your system fails, you can recover your edits by reentering the EDIT command followed by the /RECOVER qualifier. EDT recreates your last editing session on your screen up to the point where it was interrupted. It uses the contents of a journal file that is maintained during the editing session.

EDT provides an online help facility that you can access during an editing session. In line mode, you can enter the HELP command. EDT displays general information on EDT as well as detailed information on both line mode editing and nokeypad mode editing. In keypad mode, you can press the HELP key or the PF2 key. EDT displays a keypad diagram on your terminal screen, and a list of keypad editing keys. For help on a specific keypad function, press the key you want help on.

For more information on the EDT editor, see the *EDT Editor Manual*.

2.2 Compiling a BASIC–PLUS–2 Program

The primary functions of the BASIC–PLUS–2 compiler are as follows:

- Detect errors in your source program
- Generate any appropriate error messages
- Generate machine language instructions from the source statements
- Group these language instructions into an object module for the linker

In the generated object module, the BASIC–PLUS–2 language elements are replaced by *thread names*. Thread names point to segments or *threads* of code stored in the BASIC–PLUS–2 libraries. These threads perform the tasks associated with the BASIC–PLUS–2 language elements. The object module does not contain executable code but rather thread names pointing to executable code. When you link your BASIC–PLUS–2 program, you invoke the Task Builder, which replaces each thread name in the object module with the executable code it points to.

If you compile a program that contains an error, the compiler signals a compile-time error message. See the *BASIC–PLUS–2 Reference Manual* for a list of the BASIC–PLUS–2 compile-time errors and the user action required to correct them.

See the *BASIC–PLUS–2 Reference Manual* for a list of the Object Time System (OTS) routines and their corresponding thread names. See Chapter 18 for more information about BASIC–PLUS–2 libraries.

To invoke the BASIC–PLUS–2 compiler, you use the DCL command BASIC. The following sections describe the BASIC command and BASIC command qualifiers.

2.2.1 The BASIC Command

To compile your source program at DCL level, use the BASIC command. The BASIC command has the following format:

BASIC [/global-qualifier]... . . . {file-spec [/local-qualifier] . . . }, . . .

/global-qualifier
Names a qualifier, thereby indicating a specific action to be performed by the compiler on every input source file specified in the command. Global qualifiers can be overridden by local qualifiers of the same type.

/local-qualifier
Names a qualifier, thereby indicating a specific action to be performed by the compiler on the one input source file to which that qualifier has been appended. Local qualifiers can override global qualifiers of the same type.

file-spec
Specifies an input source file containing a program or module to be compiled. The BASIC–PLUS–2 compiler assumes the file to be of the default file type, .B2S, if no file type is included in the file specification.

You can specify any number of source files for compilation as long as you do not exceed maximum command length. On RSX, the command line is limited to 200 characters. On RSTS/E, the command line limit is 127 characters.

If you enter the BASIC command with no parameters, you enter the BASIC environment. For more information on the BASIC environment, see Chapter 1.

2.2.2 BASIC Command Qualifiers

The following is a list of the BASIC command qualifiers and their defaults. A description of each qualifier follows the list.

Command Qualifiers	Defaults
/[NO]BOUND	/BOUND
/[NO]BUILD	/NOBUILD
/[NO]CHAIN	See text.
/[NO]CROSS_REFERENCE [= [NO]KEYWORDS]	/NOCROSS_REFERENCE
/[NO]DEBUG	/NODEBUG
/[NO]FLAG [= [NO]DECLINING]	/FLAG=DECLINING
/[NO]LINES	/LINES
/[NO]LIST [= file-spec]	/NOLIST
/[NO]MACRO [= file-spec]	/NOMACRO
/[NO]OBJECT [= file-spec]	/OBJECT
/SCALE = int-const	/SCALE=0
/USING = file-spec	See text.

/VARIANT = int-const /VARIANT=0
/[NO]WARNINGS /WARNINGS

Command Qualifiers

/[NO]BOUND

The /NOBOUND qualifier eliminates the overhead of checking array boundaries when referencing memory-resident arrays of one or two dimensions. This can improve run time performance, but it should be used only after the user is confident that the array handling activity of the program is sound. Specifying or defaulting the /BOUND qualifier results in full array boundary checking.

CAUTION

When you specify /NOBOUND, the compiler generates array threads that omit boundary checking. If you incorrectly index beyond array limits, the OTS does not trap your errors. The consequences of such misuse are unpredictable. The user is responsible for ensuring the array handling integrity of the program before taking advantage of the /NOBOUNDS compilation option.

/[NO]BUILD

The /BUILD qualifier causes a command (CMD) file and an overlay description language (ODL) file to be generated. The CMD file contains instructions that enable the Task Builder to link your program module or modules with libraries and other routines. The ODL file specifies how program segments should be organized in memory during program execution. Normally, you need to generate these files only once for a program. The default is /NOBUILD.

/[NO]CHAIN

The /CHAIN qualifer can be used on RSTS/E systems only. The /CHAIN qualifier enables other programs to CHAIN into the program using the LINE clause of the CHAIN statement. If the program has more than 200 line numbers, the /NOCHAIN qualifier reduces the memory needs of the output program by disabling storage of line numbers in memory. You cannot chain from one DECNET node to another. The default is determined at installation.

/[NO]CROSS_REFERENCE [=[NO]KEYWORDS]

If you use the /CROSS_REFERENCE qualifier with the /LIST qualifier when you compile your program, the BASIC–PLUS–2 compiler includes cross-reference information in the program listing file. If you specify /CROSS_REFERENCE=KEYWORDS, BASIC–PLUS–2 also cross-references BASIC–PLUS–2 keywords used in the program. If you specify

/NOCROSS_REFERENCE, BASIC–PLUS–2 does not include a cross reference section in the compiler listing. The default is /NOCROSS_REFERENCE.

/[NO]DEBUG

The /DEBUG qualifier appends information on symbolic references and line numbers to the object file. This information is used by the BASIC–PLUS–2 debugger to debug your program. You must specify the /LINES qualifier when you specify the /DEBUG qualifier on the COMPILE command; otherwise, BASIC–PLUS–2 signals an error.

When you specify /DEBUG, control is passed to the debugger when the program is executed in the BASIC–PLUS–2 environment. If you specify /NODEBUG, information on program symbols and line numbers is not included in the object file and control is not passed to the debugger when the program executes. The default is /NODEBUG.

See Chapter 3 for information on the BASIC–PLUS–2 debugger.

/[NO]FLAG [= [NO]DECLINING]

The /FLAG qualifier causes BASIC–PLUS–2 to flag program elements that are not recommended for new program development. For example, if you specify the DECLINING clause, BASIC–PLUS–2 flags the following source code as declining:

- CVT$$ (use EDIT$)
- CVT$%, CVT$F, CVT%$, CVTF$, AND SWAP% (use multiple MAP statements)
- DEF* functions (use DEF functions)
- FIELD statements (use MAP DYNAMIC and REMAP)
- GOTO line-num% (do not use the integer suffix with a line number)

The default is /FLAG=DECLINING.

/[NO]LINES

The /LINES qualifier includes line number information in object modules. If you specify /NOLINES, BASIC–PLUS–2 does not include line number information in object modules. If you specify /NOLINES in a program containing the run-time ERL function, BASIC–PLUS–2 issues a warning that the /NOLINES qualifier has been overridden. The default is /LINES.

/[NO]LIST [= file-spec]
The /LIST qualifier causes BASIC–PLUS–2 to produce a compiler listing file. The name of the listing file is the name you specify or, if you do not supply a file specification, is the name of the program being compiled. The listing file has a default file type of LST. If you specify /NOLIST, BASIC–PLUS–2 does not generate a compiler listing. /NOLIST is the default.

/[NO]MACRO [= file-spec]
The /MACRO qualifier converts the program into MACRO–11 source code and saves it in a file you specify. If you do not supply a file specification, the macro source code is placed in a file with the same name as the program and a file type of MAC. A MACRO–11 file can be assembled. If you specify /NOMACRO, a MACRO–11 source code file is not generated. You cannot specify the /OBJECT qualifier with the /MACRO qualifier. The default is /NOMACRO.

/[NO]OBJECT [= file-spec]
The /OBJECT qualifier generates an object module with the file specification you specify. If you do not supply a file specification, the object file has the same name as the program and a file type of OBJ. The /NOOBJECT qualifier allows you to check your program for errors without creating an object file. If your program contains one or more fatal errors, an object module is not generated. You cannot specify the /MACRO qualifier with the /OBJECT qualifier. /OBJECT is the default.

/SCALE = int-const
The /SCALE qualifier allows control of accumulated round-off errors when double precision numbers (values typed DOUBLE) are used. Numbers are stored as multiples of 10 by setting *int-const* (the scale factor) from 0 through 6. A scale factor larger than six causes BASIC–PLUS–2 to signal the error message "Scale factor out of range–ignored." /SCALE=0 is the default.

/USING = file-spec
The /USING qualifier can be used on RSX systems only. The *file-spec* specifies the task name assigned to the BASIC–PLUS–2 compiler, if a name other than BP2 was chosen for the compiler during installation. If you do not specify the /USING qualifier, the default compiler name used is BP2.

/VARIANT = int-const
The /VARIANT qualifier establishes *int-const* as a value to be used in compiler directives. The variant value can be referenced in a lexical expression with the lexical function, %VARIANT. *Int-const* always has a data type of WORD. The default is /VARIANT=0.

/[NO]WARNINGS

The /WARNINGS qualifier causes BASIC–PLUS–2 to display warning messages during program compilation. The /NOWARNINGS qualifier causes BASIC–PLUS–2 to disable warning messages during program compilation. The default is /WARNINGS.

2.2.3 Compiler Listings

A compiler listing provides information that can help you debug your BASIC–PLUS–2 program. To generate a listing file, specify the /LIST qualifier when you compile your BASIC–PLUS–2 program interactively. For example:

```
$ BASIC/LIST MAIN.B2S
```

If the program is compiled as a batch job, the listing file is created by default; specify the /NOLIST qualifier to suppress creation of the listing file. By default, the name of the listing file is the name of the source program followed by a file type of LST. You can include a file specification with the /LIST qualifier to override this default.

The following sections contain examples of a listing file generated by the following command:

```
$ BASIC/BUILD/LIST/CROSS_REFERENCE Lister.B2S
```

The numbered explanations in each section correspond to the highlighted numbers in each example.

2.2.3.1 Source Program Listing

The source program section of the compiler listing contains the source code plus listing line numbers generated by the compiler.

```
❶ LISTER    Listing Tester                                    21-Feb-91 06:48 PM
  V2.7      Test                                              SY:LISTER

❷      00001 10    %TITLE "Listing Tester"
       00001       %SBTTL "Test"
       00001       !This program only shows the format of a listing
       00001       !file.  It does no useful work.
       00001       %IDENT "V2.7"
       00001       %INCLUDE "MAPS.DEF"
❸ I1   00001       !MAP definition file
  I1   00001       MAP (SHARED)    STRING A = 16,  &
  I1   00001                       LONG B,         &
  I1   00001                       DOUBLE C,       &
  I1   00001                       BYTE D
       00002       DECLARE INTEGER INDEX
       00003       DECLARE LONG CONSTANT TRUE = -1
       00004       DECLARE SINGLE Q(5)
       00004       %IF %VARIANT = 2
       00004       %THEN
❹    F100004               DECLARE DOUBLE Z(10)
       00004       %END %IF
       00005    First_loop:
       00005       FOR INDEX = 0 TO 5
       00006         PRINT Q(INDEX)
       00007       NEXT INDEX
       00008    Second_loop:
       00008       WHILE TRUE
       00009         INDEX = INDEX + 1
       00010         EXIT Second_loop IF INDEX =>5
       00011       NEXT
       00001 32767 END
```

Explanation:

❶ This is the listing header.

It contains

- The name of the program module
- Text specified in the %TITLE directive
- The date and time of the compilation
- Text specified in the %IDENT directive
- Text specified in the %SBTTL directive
- The file specification of the source file

❷ These are statement numbers.

This lists the number of the last statement on each line of text. BASIC uses these line and statement numbers when reporting compile-time errors. Also, these numbers help you set breakpoints on a multi-statement line when using the debugger.

❸ This is %INCLUDE file information.

The I tells you that this code was extracted from a %INCLUDE file. The number following the I tells you the depth of nested %INCLUDE directives. Because this %INCLUDE directive occurs in the source program, the number is 1. If the %INCLUDE file itself contained a %INCLUDE directive, the code extracted from that file would be numbered 2, and so on.

❹ This is a true-false flag for %IF-%THEN-%ELSE-%END-%IF directives.

Lines marked with T are compiled. Lines marked with F are not compiled.

2.2.3.2 Cross-Reference Listing

If you specified the /CROSS_REFERENCE qualifier, your listing includes a section displaying a list of the names of every identifier, both predeclared and user-declared, and every label to which the source code refers.

```
LISTER    Listing Tester                                 21-Feb-91 06:48 PM
V2.7      Test                                           SY:LISTER
❺ User Identifier Cross Reference
Symbol                                    Datatype   Storage Name Type
        +-----------------------------------------+
        !        #  Defining reference            !
        !        @  Destructive reference         !
        !        P  Parameter reference           !
        !        R  Redefining reference          !
        +-----------------------------------------+
A=16                                      Sta. Str.  Local   Variable  Map
            10.1#
B                                         Long       Local   Variable  Map
            10.1#
C                                         Double     Local   Variable  Map
            10.1#
D                                         Byte       Local   Variable  Map
            10.1#
INDEX                                     Word       Local   Variable
            10.2#           10.5              10.6            10.7
            10.9@           10.9              10.10
Q()                                       Single     Local   Array
            10.4#           10.6
TRUE                                      Long       Local   Constant
            10.3#           10.8
```

Explanation:

❺ This is the cross-reference listing for variables and named constants. This section lists

- All variable names
- The line number and statement at which variable names are referenced
- The data type of variable names
- Whether the storage for variable names is external or local to this program module
- The type of PSECT containing variable names (if any)

The symbols in the box at the top of this section are described as follows:

# Defining reference	Indicates a statement that defines a symbol.
@ Destructive reference	Indicates a statement that modifies the value of a symbol.
P Parameter reference	Indicates a symbol that is passed as a parameter to a routine; therefore, it cannot be determined if it is a destructive reference. This symbol may or may not be modified in the subroutine.
R Redefining reference	Indicates a symbol that is redimensioned or redefined by a statement.

❻
```
MAP Definition Cross Reference
Symbol                                     Datatype   Storage Name Type
SHARED                                                        Map
A=16                                       Sta. Str.  Local   Variable  Map
               10.1#
B                                          Long       Local   Variable  Map
               10.1#
C                                          Double     Local   Variable  Map
               10.1#
D                                          Byte       Local   Variable  Map
               10.1#
```

Explanation:

❻ This is the cross-reference listing for mapped variables. This section lists

- All variable names
- The line number and statement number at which variable names are referenced
- The data type of variable names

- Whether the storage for variable names is external or local to the program module
- The type of PSECT containing variable names (if any)

```
LISTER    Listing Tester                                    21-Feb-91 06:48 PM
V2.7      Test                                              SY:LISTER
❼ Label Cross Reference
Symbol                                  References
FIRST_LOOP
            10.5#
SECOND_LOOP
            10.8#         10.10
```

Explanation:

❼ This is the label cross-reference listing for labels.

This tells you the label names and the line number and statement at which they are referenced.

2.2.3.3 Qualifier Summary

The compilation summary lists the qualifiers used with the BASIC command and the compilation statistics.

```
LISTER                                                      21-Feb-91 06:48 PM
V2.7                                                        SY:LISTER

❽ PDP-11 BASIC-PLUS-2 V2.7-00 using EIS with  run support

ENVIRONMENT INFORMATION:              RMS FILE ORGANIZATION:
    Current edit line : 0                NO Index
    NO Modules loaded                    NO Relative
    NO Main module loaded                NO Sequential
                                         NO Virtual

DEFAULT DATA TYPE INFORMATION:        LISTING FILE INFORMATION:
    Data type : REAL                        Source
    Real size : SINGLE                      Cross Reference
    Integer size : WORD                  NO Keywords
    Scale factor : 0                        60 lines by 80 columns

COMPILATION QUALIFIERS:               BUILD QUALIFIERS:
       Object                            NO Dump
    NO Macro                             NO Map
       Lines                             NO Cluster
       Warnings                          NO I- and D-Space
    NO Debug records                     Task extend  : 512
    NO Syntax checking                   RMS ODL file : LB:[1,1]RMS11X
    Flag : Declining                     BP2 Disk lib : LB:[1,1]V27OTS
    Variant : 0                          BP2 Resident lib : NONE
                                         RMS Resident lib : NONE
```

8. This section lists the compiler defaults in effect when the program was compiled.

2.3 Linking a BASIC–PLUS–2 Program

You use the Task Builder to link your program. The Task Builder generates a task image file which you can run. The executable task image has a default file type of TSK.

Before linking a program, you need to generate a command (CMD) file and an overlay descriptor language (ODL) file for the program. You can generate the CMD and ODL files by using the /BUILD qualifier with the BASIC command when you compile the program. You can also generate the CMD and ODL files by using the BUILD command in the BASIC–PLUS–2 environment.

The Task Builder uses the CMD file you build for instructions on which libraries and ODL files to use, how many words to extend your task, and so on. The Task Builder uses the ODL file for instructions on how to overlay segments of your program and the libraries your program needs during the execution of your program.

Once you have compiled the program and have built CMD and ODL files, you can invoke the Task Builder by using either the TKB or LINK commands. These commands are described in the following sections.

2.3.1 The TKB Command

On RSTS/E systems, you specify the TKB command and the name of a command (CMD) file to link a program. The CMD file you specify describes which libraries the Task Builder should link with to resolve the addresses in the object file. The TKB command is also available on RSX systems through the MCR command line interface.

The TKB command has the following format:

TKB @file-spec

file-spec
A command (CMD) file generated by a previous BUILD operation. The name of the CMD file must be preceded by an at sign (@). If you do not specify a file type, the Task Builder searches for the file name you specify with a file type of CMD.

The following is an example of the TKB command:

```
$ TKB @MAIN
```

This command generates the task image file MAIN.TSK.

Note

The TKB command must be installed as a Concise Command Language (CCL) command on your system before you can use it to link a program. If it is not installed as a CCL command, you can access the Task Builder by specifying the RUN $TKB command. For more information, see the *RSTS/E Task Builder Reference Manual.*

See the *RSTS/E System User's Guide* for more information about the TKB command.

2.3.2 The LINK Command

On RSX and RSTS/E systems, you can specify the DCL command LINK to link a program. On RSTS/E systems, the LINK command uses a generalized file of commands for the Task Builder. Therefore, you do not need to build CMD and ODL files before linking. This, however, causes the Task Builder to link your program to both the BASIC–PLUS–2 and RMS–11 libraries whether your program requires them or not. This can greatly increase the size of your task. Therefore, it is recommended that you use the TKB command to link your program on RSTS/E systems.

The LINK command has the following format:

On RSX Systems:

LINK file-spec /BASIC

On RSTS/E Systems:

LINK /BASIC file-spec [,file-spec[, . . .]]

/BASIC
The /BASIC qualifier causes the system to invoke the Task Builder to link the program.

file-spec
On RSX systems, *file-spec* is a command (CMD) file generated by a previous BUILD operation. On RSTS/E systems, *file-spec* is one or more object module (OBJ) files generated during compilation. If you do not specify a file type, the Task Builder searches for a file with a file type of CMD on RSX systems, and for a file with a file type of OBJ on RSTS/E systems.

An example of the LINK command is as follows:

```
$ LINK MAIN /BASIC
```

For more information on the LINK command, see the *RSTS/E System User's Guide* or the *RSX-11M-PLUS Command Language Manual.*

2.4 Executing a BASIC–PLUS–2 Program

Once you have linked your program, you can use the DCL command RUN to execute it. The RUN command has the following format:

RUN file-spec

file-spec
The name of the file you want to run. If you do not specify a file type, the compiler searches for a file type of TSK by default.

For example:

```
$ RUN MAIN.TSK
```

During program execution, an image can generate a fatal error called an exception condition. When an exception condition occurs, BASIC–PLUS–2 displays an error message. Run-time errors can also be issued by other facilities such as the PDP–11 operating system. For a description of the BASIC–PLUS–2 run-time errors, see B.

If you compile the program with the /DEBUG qualifier, the debugger displays a prompt when you run the program. You can then either specify debugger commands or choose to continue program execution by entering the CONTINUE command. See Chapter 3 for more information on the debugger. See the *BASIC–PLUS–2 Reference Manual* for a description of the BASIC–PLUS–2 debugger commands.

For more information on the RUN command, see the *RSTS/E System User's Guide* or the *RSX-11M-PLUS Command Language Manual.*

3 Debugging Programs

Debugging is the process of eliminating the errors in logic in your programs. This chapter describes how to invoke the BASIC–PLUS–2 debugger and provides examples of using the debugger in the BASIC–PLUS–2 environment and at DCL command level.

3.1 Introduction

The debugger has a set of commands that allow you to specify breakpoints. Breakpoints temporarily halt the execution of your program, at which point you can do the following:

- Look at program errors
- Display values describing storage allocation
- Examine the contents of program variables
- Assign new values to program variables
- Display values describing file status characteristics

The debugger cannot reference external variables declared in MACRO subprograms, external constants declared in MACRO subprograms, or compile-time constants.

3.2 Invoking the Debugger

You can access the debugger by specifying the /DEBUG qualifier with any of the following commands:

- The DCL command BASIC
- The COMPILE command in the BASIC–PLUS–2 environment
- The RUN command in the BASIC–PLUS–2 environment

For example, the following RUN command invokes the debugger in the BASIC–PLUS–2 environment. The debugger responds by displaying the file name of the program module that is currently executing and its number sign (#) prompt.

```
RUN/DEBUG MYPROG
DEBUG:MYPROG
#
```

Once the number sign (#) prompt appears, you can enter debugger commands. If you enter the CONTINUE command at this point, program execution begins and continues until it is forced to pause or stop (for example, if the program prompts you for input, or an error occurs).

Whenever you enter a debugger command, you must enter the CONTINUE command to execute the debugger command you enter. For example:

```
$ BASIC/DEBUG/BUILD MYPROG
$ TKB @MYPROG
$ RUN MYPROG
DEBUG:MYPROG
# BREAK 10
# CONTINUE
# EXIT
$
```

In this example:

1. The program is compiled at DCL level with the /DEBUG qualifier, then linked and run.
2. Once the program is run, the debugger displays the file name of the currently executing program module.
3. The # prompt indicates that control has passed from your program to the debugger and that you can enter debugger commands.
4. The BREAK 10 command tells the debugger to stop execution at line 10 of MYPROG.
5. The CONTINUE command passes control to your program, resumes program execution, and executes the BREAK command.
6. The EXIT command exits you from the debugger and returns you to DCL command level.

When you compile a program you want to debug, specify the /LIST qualifier to generate a compilation listing file. The compilation listing is useful during a debug operation because it numbers each program line and each statement in a program line. These numbers can help you decide where to place breakpoints in a program.

Table 3–1 lists and describes the BASIC–PLUS–2 debugger commands.

Table 3–1 Debugger Commands

Command	Function
BREAK	Stops program execution at the first executable statement in the program module that is currently executing. Breakpoints set by any form of the BREAK command remain in effect until you disable the breakpoint with the UNBREAK command.
BREAK lin-num	Stops program execution at the line number in the program module that is currently executing. You can set multiple breakpoints by separating multiple line numbers with commas. You can specify a maximum of 10 breakpoints. Program execution stops and control passes to the debugger when BASIC–PLUS–2 executes the line number you specify.
BREAK lin-num.stat-num	Stops program execution at the line number and the number of the statement in the program module that is currently executing.
BREAK lin-num.stat-num;mod-nam	Stops program execution at the line number and the number of the statement in the module you specify.
BREAK ON {CALL,DEF,LOOP}	Stops on CALL statements, user-defined functions, or loops in the program module that is currently executing.
CONTINUE	Passes control to your program, resumes program execution, and executes BREAK and TRACE commands, if entered.
CORE	Displays the number of words in memory currently allocated for your task.
ERL	Displays the line number of the module that was executing when the last error was found.
ERN	Displays the name of the module that was executing when the last error was found.
ERR	Displays the number of the last error.
EXIT	Returns control to the BASIC–PLUS–2 environment or the keyboard monitor level.

(continued on next page)

Table 3–1 (Cont.) Debugger Commands

Command	Function
FREE	Displays the number of words in memory currently available for free space.
I/O BUFFER	Displays the number of words in memory currently allocated for I/O buffer space.
LET	Changes the contents of a variable. (You cannot create new variables.)
PRINT	Displays the contents of a variable.
RECOUNT	Displays how many characters were transferred by the last input operation.
REDIRECT terminal-name	Sends all debugger I/O to a specified terminal. The terminal must be logged out and not assigned to another user.
STATUS	Returns a word-length integer that contains information about the last opened file.
STEP	Executes the next statement in the program module that is currently executing. If you enter a carriage return at the debugger prompt (#), it acts like a STEP command.
STEP number	Executes the number of statements you specify before stopping program execution and passing control to the debugger.
STRING	Displays the number of words in memory currently allocated for dynamic string space. This value represents dynamic memory allocation, not static memory allocation.
TRACE	Displays the line number of each program line as it executes.
UNBREAK	Disables breakpoints set by any form of the BREAK command.
UNBREAK ON	Disables breakpoints set by the BREAK ON CALL, DEF, and LOOP commands.
UNTRACE	Disables the TRACE command.

See the *BASIC–PLUS–2 Reference Manual* for a complete description of the BASIC–PLUS–2 debugger commands.

3.3 Sample Debugging Session in the BASIC–PLUS–2 Environment

This section includes a sample debugging session in the BASIC–PLUS–2 environment.

```
BASIC2

OLD BAL

BASIC2

LISTNH
100     PRINT 'This program keeps track of your checking balance.'
200     INPUT 'Starting balance'; BALANCE
300     INPUT 'How many checks'; NUM.CHECKS%
400     FOR I% = 1% TO NUM.CHECKS%
500              INPUT 'Amount of check'; CHECK.AMOUNT
600              BALANCE = BALANCE - CHECK.AMOUT
700     NEXT I%
800     PRINT 'Your balance is '; BALANCE
32767   END
```

This program executes with no error messages. However, no matter how many checks you write, the balance always remains at the starting value you enter in. To find out where the problem lies, execute the program with the RUN/DEBUG command:

```
BASIC2

RUN/DEBUG

DEBUG:BAL

# BREAK 600

# CONTINUE
This program keeps track of your checking balance.
Starting balance? 500
How many checks? 1
Amount of check$ 12.50

BREAK at line 600 statement 1

# STEP

# PRINT BALANCE
```

```
 500.00
# PRINT CHECK.AMOUNT
 12.5
# CONTINUE
Your balance is 500
```

Control transfers to the debugger after BASIC–PLUS–2 executes line 600. Then you can examine the values of the program variables BALANCE and CHECK.AMOUNT, perform the subtraction using these variables, and set the BALANCE variable to the correct value. However, the program still works incorrectly. You now take a closer look at the program line performing the calculation:

```
600     BALANCE = BALANCE - CHECK.AMOUT
```

Notice that the variable CHECK.AMOUNT is misspelled in line 600. This means that BASIC–PLUS–2 subtracted a variable named CHECK.AMOUT from the variable BALANCE. Because BASIC–PLUS–2 variables are initialized to zero, line 600 did not affect the balance at all. To correct the program, re-enter line 600:

```
600     BALANCE = BALANCE - CHECK.AMOUNT
```

If you specify the compiler default SET /TYPE:EXPLICIT, which requires you to explicitly declare the data types for all variables, you solve the problem of misspelling variable names.

It may be more convenient to use line numbers in your programs while debugging them, although you can specify the number of the statement in a single line number. Once you debug your program, you can eliminate unnecessary line numbers. For example:

```
100     DECLARE INTEGER I, NUM.CHECKS, DOUBLE CHECK.AMOUNT, BALANCE
        !
        PRINT 'This program keeps track of your checking balance.'
        INPUT 'Starting balance'; BALANCE
        INPUT 'How many checks'; NUM.CHECKS
        FOR I% = 1% TO NUM.CHECKS
                INPUT 'Amount of check'; CHECK.AMOUNT
                BALANCE = BALANCE - CHECK.AMOUNT
        NEXT I%
        PRINT 'Your balance is '; BALANCE
        END
```

3.4 Sample Debugging Session at DCL Command Level

This section includes a sample debugging session from DCL level.

This sample session debugs the following BASIC–PLUS–2 programs:

- MAPS.B2S
- CREATE.B2S
- UPDATE.B2S
- ACTSUB.B2S

MAPS.B2S declares the data types and storage for program variables. CREATE.B2S, UPDATE.B2S, and ACTSUB.B2S use MAPS.B2S by using the %INCLUDE compiler directive. When you compile CREATE.B2S and UPDATE.B2S, MAPS.B2S is automatically included.

```
!       MAPS.B2S
!
! Declaring the data type and storage
! for the variables used in ACCT.RMS
!
MAP (ACCT) STRING ACCT.NAME = 50,        &
        ACCT.ACCOUNT = 6,          &
        SINGLE ACCT.BALANCE
!
! Declaring the data type and storage
! for the variables used in LEDGER.RMS
!
MAP(LEDGER) STRING LEDGER.NAME = 22,     &
        LEDGER.DATE = 6,          &
        LEDGER.ACCOUNT = 6,       &
        SINGLE LEDGER.AMOUNT
```

ACTSUB.B2S is a FUNCTION subprogram that opens two files named ACCT.RMS and LEDGER.RMS.

```
FUNCTION BYTE ACTSUB (STRING FILES.NAME, INTEGER CHANNEL_NUMBER)
!
!               ACTSUB.B2S
!
! This is the FUNCTION subprogram used to initialize the files.
! There is one parameter passed to the subprogram:
!
! FILES.NAME    This is the 1- to 6-character name of the file.
!               The file name variable returns unchanged to the main
!               program.
!
! The value returned to the programs that call this program
! is a truth flag that indicates the success or failure of a
! file open operation. The function returns a zero if the file
! open operation is successful, or a -1 if it is not.
!
! Use the maps declared in MAPS.B2S.
!
%INCLUDE 'MAPS.B2S'
!
! Truth flag:
!
ACTSUB = 0%
!
2       IF FILES.NAME = 'ACCT'
        THEN
                OPEN 'ACCT.RMS' AS FILE #CHANNEL_NUMBER, &
                        INDEXED FIXED,  &
                        ACCESS MODIFY,  &
                        ALLOW MODIFY,   &
                        MAP ACCT,       &
                        PRIMARY ACCT.NAME,      &
                        ALTERNATE ACCT.ACCOUNT  DUPLICATES CHANGES
        ELSE
            IF FILES.NAME = 'LEDGER'
                THEN
                        OPEN 'LEDGER.RMS' AS FILE #CHANNEL_NUMBER, &
                            INDEXED FIXED,      &
                            ACCESS MODIFY,      &
                            ALLOW MODIFY,       &
                            MAP LEDGER,         &
                            PRIMARY LEDGER.ACCOUNT DUPLICATES,   &
                            ALTERNATE LEDGER.DATE DUPLICATES CHANGES
                ELSE
                        ! File open not successful
                        ACTSUB = -1%
                        PRINT FILES.NAME;' Undefined in ACTSUB'
            END IF
END IF
END FUNCTION
```

CREATE.B2S calls ACTSUB.B2S to open the accounting and ledger files, asks the user for information about the account, writes the information to the open files, and finally closes the files.

```
1       !       CREATE.B2S
        !
        ! Use the variables declared in MAPS.B2S
        !
        %INCLUDE 'MAPS.B2S'
        !
        ! Declare the FUNCTION subprogram
        !
        EXTERNAL BYTE FUNCTION ACTSUB (STRING,INTEGER)
        !
        DECLARE INTEGER FILE.STATUS !FILE.STATUS is the truth flag
 Open:  !
        ! Open the files
        !
        ! Call ACTSUB to open ACCT.RMS
        !
        FILE.STATUS = ACTSUB ('ACCT',1%)
        !
        ! Call ACTSUB to open LEDGER.RMS
        !
        FILE.STATUS = ACTSUB ('LEDGER',2%)
        !
 Ask.Acct: !
           ! Enter account records
           !
           LINPUT 'Account name';ACCT.NAME
           !
           ! If the user enters a null string,
           ! close the files and exit the program
           !
           GOTO Ask.Ledger IF ACCT.NAME = ''
           !
           LINPUT 'Account number';ACCT.ACCOUNT
           INPUT 'Account balance';ACCT.BALANCE
           !
           PUT #1 ! Write these three records to ACCT.RMS
           !
           GOTO Ask.Acct ! Loop to enter more records
           !
```

```
Ask.ledger: !
            ! Enter ledger records
            !
             LINPUT 'Ledger name';LEDGER.NAME
            !
            ! If the user enters a null string,
            ! close the files and exit the program
            !
            GOTO Done IF LEDGER.NAME = ''
            !
            LINPUT 'Ledger account';LEDGER.ACCOUNT
            LINPUT 'Ledger date';LEDGER.DATE
            INPUT 'Ledger amount';LEDGER.AMOUNT
            !
            PUT #2% ! Write these records to LEDGER.RMS
            !
             GOTO Ask.Ledger ! Loop to enter more records
            !
Done:   CLOSE #1,#2
        END
```

UPDATE.B2S calls ACTSUB.B2S and allows the user to update records in the account and ledger files. This program handles three different error conditions:

- If the record is locked, the program waits one second and then tries to retrieve the record again.
- If the record cannot be found, a message is displayed.
- If any other errors occur, the name of the program, the line number where the error occurred, and the error message for that error are displayed.

```
1       !       UPDATE.B2S
        !
        ! Use the variables declared in MAPS.B2S
        !
        %INCLUDE 'MAPS.B2S'
        !
        ! Declare the FUNCTION subprogram
        !
        EXTERNAL BYTE FUNCTION ACTSUB (STRING,INTEGER)
        !
        ! Declare the data type and storage for
        ! this program's variables
        !
        DECLARE INTEGER FILE.STATUS, &
                CHANGE.ACCT.RECORD
        !
        MAP (FOO) STRING OLD.ACCOUNT= 6, &
                NEW.ACCOUNT= 6, &
                ANSWER = 1
```

```
        !
        ! In the event of an error, transfer control to line 4
        !
        ON ERROR GOTO 4
        !
        ! If the open operation was not successful,
        ! close ACCT.RMS
        !
        IF ACTSUB ('ACCT',1) = -1%
                THEN
                    GOTO 5
        END IF
        !
        ! If the open operation was not successful,
        ! close LEDGER.RMS
        !
        IF ACTSUB ('LEDGER',2) = -1%
                THEN
                    GOTO 5
        !
2       !
        ! Ask for the old account numbers
        !
        INPUT 'Account number to change';OLD.ACCOUNT
        !
        ! If the user enters a null string,
        ! close the files and exit the program
        !
        IF OLD.ACCOUNT = ''
                THEN
                    GOTO 5
        END IF
        !
        ! Enter new account number
        !
        INPUT 'New account number';NEW.ACCOUNT
        !
        ! Change this record in LEDGER.RMS too?
        !
        INPUT 'Change ledger record too';ANSWER
        CHANGE.ACCT.RECORD = (ANSWER = 'Y')
        !
        ! Retrieve the ACCT.RMS record
        ! Signal an error if the record is not found
        !
```

```
3       GET #1, KEY #1 EQ OLD.ACCOUNT
        !
        ! If ANSWER is Y, then change the
        ! account number record in both files
        !
        GOTO Get2 IF NOT CHANGE.ACCT.RECORD
        !
        ! Change the old account number to
        ! the new account number
        !
        ACCT.ACCOUNT = NEW.ACCOUNT
        !
        ! Change the old record in ACCT.RMS
        !
        ! UPDATE #1
        !
        ! Retrieve the LEDGER.RMS record
        !
 Get2:  GET #2, KEY #0 EQ ACCT.ACCOUNT
        !
        ! Change the old account number to
        ! the new account number
        !
        LEDGER.ACCOUNT = NEW.ACCOUNT
        !
        DELETE #2 ! Delete the old record
        !
        PUT #2 ! Write the new record
        !
        GOTO 2 ! Loop to change more records
        !
```

```
4 Errs: SELECT ERR
        CASE 19,154
                !
                ! If the record is interlocked, then count the number
                ! of times the error occurs.  Every fifth time the error
                ! occurs, print a message, wait for one second,
                ! then try to perform the record operation again.
                !
                INTERLOCK.COUNT = INTERLOCK.COUNT +1
        PRINT 'Record is locked -- will try again'
                IF INTERLOCK.COUNT = INTERLOCK.COUNT / 5% * 5%
                        THEN SLEEP 1%
                        RESUME
                END IF
                !
        CASE 155
                ! If the record you specify does not exist,
                ! print the message:
                !
        PRINT 'That account number is not in the file'
        RESUME
                !
        CASE ELSE
                ! If any other error occurs, print the error text
                ! associated with the error code, the line
                ! number where the last error occurred, and the
                ! name of the module where the last error occurred.
                !
                PRINT ERT$(ERR);' at line ';ERL;' in ';ERN
                RESUME 5
        END SELECT
        !
5       !
        ! Close all open files.
        !
        CLOSE #1,#2
        END
```

To use the debugger on CREATE.B2S, compile, link, and run the program by entering the following sequence of commands:

```
$ BASIC/DEBUG/BUILD CREATE.B2S
$ BASIC/DEBUG/BUILD ACTSUB.B2S
$ TKB @CREATE
$ RUN CREATE
```

The debugging session for CREATE.B2S follows:

```
DEBUG:CREATE

#
```

The BREAK command sets a breakpoint that tells the debugger to stop execution at line 1, statement 2, in the FUNCTION subprogram named ACTSUB.B2S. The CONTINUE command tells the debugger to resume program execution.

```
# BREAK 1.2;ACTSUB
```

```
# CONTINUE
```

When BASIC–PLUS–2 reaches line 1, statement 2, in the subprogram, control passes to the debugger, and BASIC–PLUS–2 then displays a message telling you where execution has stopped and then displays another prompt:

```
BREAK at line 1, statement 2, in SUB:ACTSUB
#
```

At this point in the program, the files ACCT.RMS and LEDGER.RMS have not been opened. The CORE command displays the number of words in memory currently allocated for your task:

```
# CORE
CORE = 9247
```

The I/O BUFFER command tells you how much storage is allocated by the I/O buffer. A value of zero indicates that no files have been opened:

```
# I/O BUFFER
I/O BUFFERS =  0
```

The STRING command tells you how many words are allocated to dynamic string storage and not static storage:

```
# STRING
STRING SPACE =  106
```

The FREE command tells you how many words are available for I/O and dynamic string operations. Generally speaking, if your program performs several I/O operations and the value of the FREE command is less than 20 words, you should pre-extend the size of your task by specifying the EXTTSK option in the Task Builder command file. See the *RSTS/E RMS–11 MACRO Programmer's Guide* or the *RSX–11M/M–PLUS RMS–11 Macro Programmer's Guide* for more information on the EXTTSK option. The *RSTS/E Task Builder Reference Manual* or the *RSX–11M/M–PLUS Task Builder Reference Manual* also contains information about the EXTTSK option.

```
# FREE
FREE SPACE =  989
```

The UNBREAK command disables the breakpoint at line 1, statement 2, in ACTSUB.B2S:

```
# UNBREAK
```

The BREAK command sets another breakpoint after the files in the subprogram are opened:

```
# BREAK 1.8;ACTSUB
# CONTINUE
```

After BASIC–PLUS–2 executes statements 2 through 8, control passes to the debugger. The debugger displays a message telling you where execution has stopped and another prompt:

```
BREAK at line 1, statement 8, in SUB:ACTSUB

#
```

Enter the CORE, I/O BUFFER, STRING, and FREE commands again to see how the memory allocation for your program changes:

```
# CORE
CORE =  10047

# I/O BUFFER
I/O BUFFERS =  1776

# STRING
STRING SPACE =  110

# FREE
FREE SPACE =  9
```

To make sure the files are open, use the PRINT command to return the value of FILE.STATUS. A value of 0 indicates that the files are open:

```
# PRINT FILE.STATUS
0
```

The STEP command executes one statement at a time:

```
# STEP

STEP at line 1, statement 7

Account name? Reilly

# CONTINUE

Account number? 123456
Account balance? 560.00
```

At this point in the program, these records should be written to ACCT.RMS. The RECOUNT command displays how many characters were tranferred by the last file input operation:

```
# RECOUNT

18
```

The CONTINUE commands resumes program execution:

```
# CONTINUE

Account name?
Ledger name? Reilly
Ledger account? 123456
Ledger date? 6-30-82
Ledger amount? 45.60
Ledger name?
```

CREATE.B2S appears to be working as expected.

To use the debugger on UPDATE.B2S, enter the following sequence of commands:

```
BASIC

PDP-11 BASIC-PLUS-2 V2.7-00

BASIC2

OLD UPDATE

BASIC2

COMPILE/DEBUG

BASIC2

OLD ACTSUB

BASIC2

COMPILE/DEBUG

BASIC2

BUILD UPDATE,ACTSUB

BASIC2

EXIT
$ TKB @UPDATE
```

The debugging session for UPDATE.B2S follows:

```
RUN UPDATE
```

The debugger begins the debugging session by displaying the name of the program module it is currently debugging and its prompt (#):

```
DEBUG: UPDATE

#
```

The BREAK command sets a breakpoint at line 2, statement 1, in UPDATE.B2S:

```
# BREAK 2.1;UPDATE

# CONTINUE
```

When BASIC–PLUS–2 reaches line 2, statement 1, in UPDATE.B2S, control passes to the debugger:

```
BREAK at line 2, statement 1
#
```

The TRACE command displays the line number of each program line as it executes:

```
# TRACE
# CONTINUE

at line 2, statement 1

Account number to change? 12345

at line 2, statement 2

at line 2, statement 3

at line 2, statement 4

at line 2, statement 5

at line 2, statement 6

New account number? 234567

at line 2, statement 7

Change ledger record too?
```

It is not clear whether the user should answer the previous question with N, Y, NO, or YES. You should change the question in the program to supply the acceptable choices. For example:

```
INPUT 'Change ledger record too <Y/N>';ANSWER

Change ledger record too <Y/N>? Y

at line 2, statement 8

at line 3, statement 1

at line 4, statement 9

at line 4, statement 10

That account number is not in the file

at line 4, statement 11

at line 3, statement 1
```

```
at line 4, statement 9
at line 4, statement 10
That account number is not in the file
at line 4, statement 11
at line 3, statement 1
```

Ctrl/C

Note that the account number to change was incorrect. When the RESUME statement in line 4, statement 11, transferred control to the program line (line 3) that caused the error, an infinite loop was created. The Ctrl/C aborted the debugging session.

This debugging session tells you the following:

1. The files in the subprogram were opened and the records were written to them.
2. Line 2, statement 7, should be rewritten to be self-documenting.
3. Line 4, statement 11, should be rewritten to:

   RESUME 2

3.5 Debugger Error Messages

This section lists the error messages for the BASIC–PLUS–2 debugger and their possible causes.

?What?

Explanation: The debugger does not understand the command.

User Action: Check the spelling, syntax, and validity of the command.

%Bad line spec in (UN)BREAK

Explanation: You specified a nonexistent line, used incorrect syntax in specifying a program or subprogram, or used incorrect syntax when listing multiple breakpoints.

User Action: Check the syntax and change the BREAK or UNBREAK command.

%Can't CONTINUE or STEP

Explanation: The program encountered an error it could not handle and execution stopped.

User Action: Enter EXIT to exit from the debugger. Program execution cannot resume.

%Cannot open device

Explanation: You specified, for the REDIRECT command, a device that does not exist or cannot be used.

User Action: Change the device specification to one that is available.

%Data error in LET or PRINT

Explanation: The debugger encountered a program conversion error while processing a LET or PRINT command.

User Action: Change the LET or PRINT command format.

%Illegal syntax in LET

Explanation: The format of the LET debugger command is incorrect; you tried to create a new variable or assigned an expression to the variable.

User Action: Change the LET command. You cannot create a new variable with the LET command. You can specify a constant or variable as the new value, but not an expression. The LET command allows you to test or change only one variable at a time. To change or test multiple variables, use multiple LET commands.

%Illegal syntax in PRINT

Explanation: Either the format of the PRINT debugger command is incorrect, or you tried to print a variable not in the currently executing module.

User Action: Change the PRINT command. You cannot print a variable which does not exist in the currently executing program module. The PRINT command allows you to display only one variable at a time. To display multiple variables, use multiple PRINT commands.

%No room

Explanation: You specified too many breakpoints for a BREAK or UNBREAK command.

User Action: None. The debugger accepts as many as ten breakpoints and ignores the rest.

%On error entry in debugger

Explanation: A Ctrl/C trap or program error has stopped program execution and the debugger.

User Action: None.

%Stop at line N in subprogram X

Explanation: A STOP statement in the program halted program execution at line <n> in the subprogram named <X>(Debugger<XS>error messages).

User Action: Enter CONTINUE to resume execution.

4
Program Elements

A BASIC–PLUS–2 program is a series of instructions for the BASIC–PLUS–2 compiler. These instructions, no matter how varied, are all built using the same fundamental elements of BASIC–PLUS–2. This chapter describes the elements or building blocks of BASIC–PLUS–2.

4.1 Line Numbers

A BASIC–PLUS–2 program is a series of instructions for the BASIC–PLUS–2 compiler. These instructions are in the form of BASIC–PLUS–2 statements. BASIC–PLUS–2 programs require at least one line number at the first statement of the program.

When you use line numbers, you must follow these rules:

- A line number must be a unique integer between 1 and 32767. If your program has duplicate line numbers, the last line with that number replaces the previous one.
- A line number must begin in the first character position on the line.
- A line number can contain leading zeros; however, embedded spaces, tabs, and commas are invalid in line numbers.
- There must be a line number on the first line of the program.
- There can be a maximum of 32767 characters associated with a single line number.

Although line numbers on every line are not required, you may want to use them on every line that could cause a run-time error, depending on the type of error handling you use. See Chapter 15 for more information about handling run-time errors.

4.2 Labels

A label is a 1- to 31-character identifier that you use to identify a block of statements. All label names must begin with a letter; the remaining characters, if any, can be any combination of letters, digits, dollar signs ($), underscores (_) or periods (.). Labels cannot begin in column one.

When you use a label to mark a program location, you must end the label with a colon (:). The colon is used to show that the label name is being defined instead of referenced. When you reference the label, do not include the colon. In the following example, the label names end with colons when they mark a location, but the colons are not present when the labels are referenced.

Example

```
10  OPTION TYPE = EXPLICIT         ! Require declarations
    DECLARE INTEGER A, B
   .
   .
   .
    Outer_decision:
          IF A <> B
          THEN
    Inner_decision:
            IF B = C
            THEN
                A = A + 1%
                GOTO Outer_decision
            ELSE
                B = B + 1%
                GOTO Inner_decision
            END IF
          END IF
```

Labels have no effect on the order in which program lines are executed; they are used to identify a statement or block of statements.

4.3 Continuing Long Program Statements

If a program line is too long for one line of text, you can continue the program line by typing an ampersand (&) and pressing the RETURN key. Note that only spaces and tabs are valid between the ampersand and the carriage return.

A single statement that spans several text lines requires an ampersand at the end of each continued line.

Example

```
20  OPEN "SAMPLE.DAT" AS FILE #2%,        &
          SEQUENTIAL VARIABLE,            &
          RECORDSIZE 80%
```

In an IF...THEN...ELSE construction, statement separators are not necessary. If a continuation line begins with THEN or ELSE, then no statement separator is necessary. Similarly, in a line following a THEN or ELSE, there is no statement separator.

Example

```
200  IF (A$ = B$)
     THEN
        PRINT "The two values are equal"
     ELSE
       PRINT "The two values are different"
     END IF
```

Several statements can be associated with a single program line. If there are several statements on one line, they must be separated by backslashes (\ ).

Example

```
300  PRINT A \ PRINT V \ PRINT G
```

Because all statements are on the same program line, any reference to this program line refers to all three statements. In the preceding example, BASIC–PLUS–2 cannot execute just one of the statements without executing the other two as well.

4.4 Identifying Program Units

You can delimit a main program compilation unit with the PROGRAM and END PROGRAM statements. This allows you to identify a program with a name other than the file name. The program name must not duplicate the name of a SUB or FUNCTION subprogram.

```
20  PROGRAM Sort_out
    .
    .
    .
    END PROGRAM
```

If you include the PROGRAM statement in your program, the name you specify becomes the module name of the compiled source. This feature is useful when you use object libraries because the librarian stores modules by their module name rather than the file name. Similarly, module names are used by the BASIC–PLUS–2 debugger and the Task Builder.

4.5 The BASIC–PLUS–2 Character Set

BASIC–PLUS–2 uses the full ASCII character set, which includes the following:

- The letters A through Z, both upper- and lowercase
- The digits 0 through 9
- Special characters

See C for a complete list of the ASCII character set and character values.

The BASIC–PLUS–2 compiler does not distinguish between upper- and lowercase letters; exceptions are letters inside quotation marks (called *string literals*) and letters in a DATA statement. The compiler also does not process characters in a REM statement or comment field.

You can use nonprinting characters in your program, for example, in string literals and constants, but to do so you must do one of the following:

- Use a predefined constant such as ESC or DEL
- Use the CHR$ function to specify an ASCII value
- Use a character literal such as "ASCII code" C

See Section 4.8 for more information on predefined constants. See Chapter 8 for more information on the CHR$ function.

4.6 Program Documentation

Documenting a program is the process of mixing text (comments) and code in a way that helps make the program more understandable. Program documentation does not affect the way a program executes.

You can sprinkle comments liberally throughout a program; however, programs that are neatly structured need fewer comments. You can clarify your code by following these suggestions:

- Use meaningful variable names
- Include sufficient white space
- Indent your program lines according to the structure of your code

In BASIC–PLUS–2, there are two ways to include comments in a program: the comment field (!) and the REM statement.

A comment field starts with an exclamation point (!) and ends with a carriage return or another exclamation point. The following example contains both comments and program statements.

Example

```
10  DECLARE                                              &
        INTEGER                                          &
        Print_page,         ! Current page number        &
        Print_line,         ! Current line number        &
        Print_column        ! Current column number
```

In this example, BASIC–PLUS–2 ignores all text between the exclamation point and the carriage return, with one exception: BASIC–PLUS–2 still recognizes the ampersand as a continuation character. Only spaces and tabs are valid between the ampersand and the carriage return.

Note

You can also terminate a comment field with an exclamation point. However, because BASIC–PLUS–2 treats any text that follows the second exclamation point as part of your program code, this practice is not recommended.

A REM statement begins with the REM keyword and ends when BASIC–PLUS–2 encounters a new line number. The text you supply between the REM keyword and the next line number documents your program. Like comment fields, REM statements do not affect program execution.

Example

```
10  REM This is a comment
20  A=5
    B=10
    REM A equals 5, B equals 10
30  C=15
```

The REM statement is nonexecutable. When you transfer control to a REM statement, BASIC–PLUS–2 executes the next executable statement that lexically follows the referenced statement. Because BASIC–PLUS–2 treats all text between the REM statement and the next line number as commentary, REM should be used very carefully in programs that follow the implied continuation rules.

4.7 Declarations and Data Types

BASIC–PLUS–2 offers two different methods for creating variables and specifying their data types:

- Implicit data typing
- Explicit data typing

With implicit data typing, BASIC–PLUS–2 creates and specifies a data type for a variable the first time you reference it in your program. With explicit data typing, you must use one of four declarative statements to name and type your program values.

BASIC–PLUS–2 has four data types:

- Integer (INTEGER)
- Floating-point (REAL)
- String (STRING)
- Record File Address (RFA)

Within the INTEGER and REAL data types there are further subdivisions: BYTE, WORD, or LONG for INTEGER and SINGLE or DOUBLE for REAL. Choosing one of these *subtypes* lets you control the following:

- The amount of storage required for the value; its container size
- The range and precision that the value can accept

For more information about data types, see Chapter 7.

4.7.1 Implicit Data Typing

With implicit data typing, BASIC–PLUS–2 creates and specifies a data type for a variable the first time you reference it. You specify the data type of the variable by a suffix on the variable name:

- A percent sign suffix (%) specifies the INTEGER data type.
- A dollar sign suffix ($) specifies the STRING data type.
- Any other ending character specifies a variable of the default data type.

The BASIC–PLUS–2 default data type is SINGLE; however, you can specify your own default with an initialization file, inside the BASIC environment, or with the OPTION statement in your program. For more information on establishing default data types, see Chapter 7 or the description of the OPTION statement in the *BASIC–PLUS–2 Reference Manual.*

The first time BASIC–PLUS–2 references one of these variables, it creates a variable with that name and data type and allocates storage for that variable.

In the following example, BASIC–PLUS–2 creates two INTEGER variables, *A%* and *B%*. Even though the values assigned to these variables are REAL, BASIC–PLUS–2 converts these values to INTEGER to match the data type specified for the variables. The sum of these two values is therefore 30, not 30.6 as it would be if the variables were named simply *A* and *B*.

Example

```
200  A% = 10.1
     B% = 20.5
     PRINT A% + B%
```

Output

```
30
```

With explicit data typing, you use a declarative statement to name and specify a data type for your program values.

4.7.2 Explicit Data Typing

BASIC–PLUS–2 has four declarative statements. These statements create variables and allocate storage. The statements are as follows:

- DECLARE
- DIMENSION
- COMMON
- MAP

The statement you choose depends on the way in which you will use the variables:

- DECLARE and DIMENSION allocate dynamic storage for variables; storage is allocated when the program executes.
- COMMON and MAP statements allocate storage for variables statically; storage is allocated when the program is compiled.

The difference between these types of storage is most apparent in the case of strings; string variables created with DECLARE can change their length during program execution, while those created with MAP and COMMON remain fixed in length. All four declarative statements associate a data type with a variable. For more information, see Chapter 7.

4.8 Constants

A constant is a value that does not change during program execution. Constants can be either literals or named constants, and can be of any data type except RFA. You can use the DECLARE CONSTANT statement to create named constants. Constants can be of the following types:

- Integer
- Floating-point

- String

In addition, BASIC–PLUS–2 provides predefined constants that are useful for:

- Formatting program output to improve clarity
- Making source code easier to understand
- Using nonprinting characters without having to look up their ASCII values

Table 4–1 lists all of the BASIC–PLUS–2 predefined constants.

Table 4–1 Predefined Constants

Constant	Decimal ASCII Value	Purpose
BEL (Bell)	7	Sounds the terminal bell
BS (Backspace)	8	Moves cursor one position to the left
HT (Horizontal Tab)	9	Moves cursor to the next horizontal tab stop
LF (Line Feed)	10	Moves cursor to the next line
VT (Vertical Tab)	11	Moves cursor to the next vertical tab stop
FF (Form Feed)	12	Moves cursor to the start of the next page
CR (Carriage Return)	13	Moves cursor to the beginning of the current line
SO (Shift Out)	14	Shifts out for communications networking, screen formatting, and alternate graphics
SI (Shift In)	15	Shifts in for communications networking, screen formatting, and alternate graphics
ESC (Escape)	27	Marks the beginning of an escape sequence
SP (Space)	32	Inserts one blank space in program output
DEL (Delete)	127	Deletes the last character entered
PI	None	Represents the number PI with the precision of the default floating-point data type

These predefined constants simplify the task of using nonprinting characters in your programs. For example, the following statement causes a bell to sound on your terminal.

Example

```
200  PRINT BEL
```

The following example prints and underlines a word on a hard copy terminal.

Example

```
300  PRINT "NAME:" + BS + BS + BS + BS + BS + "_____"
```

To print and underline the same word on a VT100 series video display terminal, use the program code in the following example. Note that the "m" must be lowercase.

Example

```
400  PRINT ESC + "[4mNAME:" + ESC + "[0m"
```

You can also create your own predefined constants with the DECLARE CONSTANT statement. The following statements define and print a constant that prints and underlines the string "NAME:".

Example

```
10  DECLARE STRING CONSTANT Underlined_word = ESC + "[4mNAME:" + ESC + "[0m"
    PRINT Underlined_word
```

Output

```
NAME:
```

For more information on constants, see Chapter 7 and the *BASIC–PLUS–2 Reference Manual*.

4.9 Variables

A variable is a unique storage location that is referred to by a variable name. The most important property of variables is that their values can change during program execution. Each named location can hold only one value at a time.

A variable name can have up to 31 characters. The name must begin with a letter; the remaining characters, if any, can be any combination of letters, digits, dollar signs ($), underscores (_), and periods (.).

Variables can be grouped in an orderly series under a single name. These groups are called *arrays*. You refer to a single variable in an array by using one or more *subscripts* that specify the variable's position in the array.

4.9.1 Floating-Point Variables

A floating-point variable is a named location that stores a single floating-point value. The storage space required to hold the value depends on the variable's REAL subtype. For example, each SINGLE floating-point variable requires 32 bits (4 bytes) of storage, while each DOUBLE floating-point variable requires 64 bits (8 bytes) of storage.

Note that if any integer value is assigned to a floating-point variable, BASIC–PLUS–2 converts the value to a floating-point number.

4.9.2 Integer Variables

An integer variable is a named location that stores a whole number. The storage space required to hold the value depends on the variable's INTEGER subtype. For example, each BYTE integer variable requires 8 bits (1 byte) of storage, each WORD integer variable requires 16 bits (2 bytes), of storage, and each LONG integer variable requires 32 bits (4 bytes) of storage.

If you assign a floating-point value to an integer variable, BASIC–PLUS–2 truncates the fractional portion of the value; it does not round to the nearest integer. In the following example, BASIC–PLUS–2 assigns the value –5 to the integer variable, not –6.

Example

```
10  B% = -5.7
```

4.9.3 String Variables

Unlike some of the numeric variables described so far, a string variable does not correspond to a single location in memory because a string variable is more likely to exceed a single location in memory. Therefore, the value of a string variable may be contained in any number of memory locations. A string variable can contain a maximum of 32767 characters; however, a string variable is still referred to by a single name as shown in the following example.

Example

```
10  DECLARE STRING Employee_name
```

4.9.4 Subscripted Variables

A subscripted variable is a floating-point, integer, RFA, or string variable that is part of an array. Chapter 10 describes arrays in detail.

An array is a set of data organized in one or more dimensions. A one-dimensional array is called a *list* or *vector*. A two-dimensional array is called a *matrix*. BASIC–PLUS–2 arrays can have up to 8 dimensions.

When you create an array, its size is determined by the number of dimensions and the maximum size, called the *bound*, of each dimension. Subscripts begin by default with 0, not 1. That is, when calculating the number of elements in a dimension, you count from zero to the bound specified. Valid array subscripts can be in the range from 0 through 32767.

The following DECLARE statement creates an 11 by 11 array of integers (11 by 11 rather than 10 by 10, because BASIC–PLUS–2 arrays are zero-based). Therefore, the array contains a total of 121 array elements.

Example

```
10  DECLARE INTEGER My_array (10, 10)
```

Note

By default, the compiler signals an error if a subscript is larger than the allowable range. Also, the amount of storage that the system can allocate depends on available memory. Therefore, very large arrays may cause an internal allocation error.

4.9.5 Initialization of Variables

BASIC–PLUS–2 sets variables to zero or null values at the start of program execution; that is, it *initializes* them. Variables initialized by BASIC–PLUS–2 include the following:

- Numeric variables and array elements (except those in MAP or COMMON statements).
- Numeric variables and array elements in a MAP statement referenced in an OPEN statement.
- String variables and array elements (except those in MAP or COMMON statements).
- Variables in subprograms. Subprogram variables (except those in MAP or COMMON statements) are initialized to zero or the null string each time the subprogram is called. Variables in a map are initialized to zero when the OPEN statement executes.
- Arrays created with an executable DIMENSION statement. BASIC–PLUS–2 reinitializes the array each time the array is redimensioned.

4.10 Keywords and Reserved Words

Keywords are elements of the BASIC–PLUS–2 language. Keywords that are not reserved can be used as user identifiers such as labels, variable or constant names, or names of MAP or COMMON areas. Depending upon the location of the keyword in your program statement, the compiler will treat it as either a keyword or a user identifier. Your BASIC–PLUS–2 programs use keywords and reserved words to do the following:

- Define data
- Perform operations

- Invoke functions

See the *BASIC–PLUS–2 Reference Manual* for a list of BASIC–PLUS–2 keywords and reserved words.

Keywords determine whether the statement is executable or nonexecutable. Executable statements such as PRINT, GOTO, and READ perform operations. Nonexecutable statements such as DATA, DECLARE, and REM, describe the characteristics and arrangement of data, usage information, and comments.

Every statement except LET and empty statements (lines that start with an exclamation point) must begin with a keyword. A BASIC–PLUS–2 keyword cannot have embedded spaces or be split across lines of text. There must be a space or tab between the keyword and any other variables or operators.

There are also phrases of keywords. In this case, the spacing requirements vary.

4.11 Operands, Operators and Expressions

An *operand* is anything that contains a value. An operand can be a scalar, a subscripted variable, a named constant, a literal, and so on. An *operator* specifies a procedure to be carried out on one or more operands. An *expression* consists of operands separated by operators.

BASIC–PLUS–2 has four types of operators:

- Arithmetic
- String
- Relational
- Logical

When combined with operands, these operators can produce the following:

- Numeric expressions
- String expressions
- Conditional expressions

For more information about operands, operators, and expressions, see the *BASIC–PLUS–2 Reference Manual.*

4.12 Assignment Statements

BASIC–PLUS–2 has statements that assign values to variables. These statements are as follows:

- LET
- INPUT
- LINPUT
- INPUT LINE
- LSET
- RSET

LET and INPUT statements allow you to assign values to any type of variable, while LINPUT and INPUT LINE allow you to assign values to string variables only.

Example

```
10  LET A = 1.25
```

LET is an optional keyword. You can assign a value to more than one variable at a time, although this is not recommended. Instead, you should use a separate assignment statement each time you assign a value to a variable.

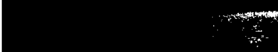
Whenever you assign a value to a numeric variable, BASIC–PLUS–2 converts the value to the data type of the variable. If you assign a floating-point value to an integer variable, BASIC–PLUS–2 truncates the value at the decimal point. If you assign an integer value to a floating-point variable, BASIC–PLUS–2 converts the value to floating-point format.

You can also assign values to variables with the DATA and READ statements; however, this method requires that you know all input data values while you are coding your program.

The INPUT, LINPUT, and INPUT LINE statements all assign values in the context of data being read into the program. These statements are discussed in Chapter 5.

The LSET statement assigns left-justified data to a string variable, while the RSET statement assigns right-justified data to a string variable. These statements are discussed in Chapter 9.

For more information on the BASIC–PLUS–2 assignment statements, see the *BASIC–PLUS–2 Reference Manual*.



5 Simple Input-Output

This chapter explains how to use the BASIC–PLUS–2 statements that move data to and from your program.

5.1 Program Input

BASIC–PLUS–2 programs receive data in three ways:

- You can enter data interactively while the program executes. You do this with INPUT, INPUT LINE, and LINPUT statements.
- If you know all the information your program will require, you can enter it as you write the program. You do this with READ, DATA, and RESTORE statements, or you can name constants with the known values.
- You can read data from files outside the program. You do this with INPUT #, INPUT LINE #, and LINPUT # statements.

The following sections describe program input in detail.

5.1.1 Providing Input Interactively

The INPUT, INPUT LINE, and LINPUT statements prompt you for data while the program executes. These statements are described in the following sections.

5.1.1.1 The INPUT Statement

The INPUT statement interactively prompts you for data. You can use the optional prompt string to clarify the input request by specifying the type and number of data elements required by the program. This is especially useful when the program contains many variables, or when someone else is running your program.

Example

```
10  INPUT "PLEASE TYPE 3 INTEGERS" ;B% ,C% ,D%
    A% = B% + C% + D%
    PRINT "THEIR SUM IS"; A%
    END
```

Output

```
PLEASE TYPE 3 INTEGERS? 25,50,75
THEIR SUM IS 150
```

When your program executes, BASIC–PLUS–2 stops at each INPUT, LINPUT, or INPUT LINE statement, prints a string prompt, if specified, and an optional question mark (?) followed by a space, and waits for input.

You can determine the format of the string prompt by using either a comma (,) or semicolon (;) on the program line:

- If you have a semicolon separating the input prompt string from the variable, BASIC–PLUS–2 prints the question mark and space immediately after the input prompt string.
- If you have a comma separating the input prompt string from the variable, BASIC–PLUS–2 prints the input prompt string, skips to the next print zone, and then prints the question mark and space.

See Section 5.2.1 for more information about print zones. For more information on formatting string prompts, see Section 5.1.1.3.

You must provide one value for each variable in the INPUT request. If you do not provide enough values, BASIC–PLUS–2 prompts you again.

Example

```
10  INPUT A,B
20  END
```

Output

```
? 5
? 6
```

BASIC–PLUS–2 interprets a carriage return (null input) as a zero value for numeric variables and as a null string for string variables.

Example

```
? 5
? [RET]
```

These responses assign the value 5 to variable *A* and zero to variable *B*. In contrast, if you provide more values than there are variables, BASIC–PLUS–2 ignores the extra values.

In the following example, BASIC–PLUS–2 ignores the extra value (8). Note that you can enter multiple values if you separate them with commas. Because commas separate variables in the PRINT statement, BASIC–PLUS–2 prints each variable at the start of a print zone.

Example

```
10  INPUT A,B,C
15  PRINT A,B,C
20  END
```

Output

```
? 5,6,7,8
 5            6             7
```

Note that this program is written to handle only the first three input values; therefore, the fourth value (8) is ignored.

If you specify a numeric variable in an INPUT statement, you must supply numeric data. The number you specify must be within the proper range for the variable's data type, or BASIC–PLUS–2 signals the error "Illegal number" (ERR=52). If you supply string data to a numeric variable or a number containing non-numeric data, BASIC–PLUS–2 signals the error "Data format error" (ERR=50). This error is also signaled if you input an integer variable containing a percent sign (%) suffix, or input a floating-point number for an integer variable.

If you specify a string variable in an INPUT statement, you can supply either numbers or letters, but BASIC–PLUS–2 treats the data you supply as a string. Because digits and a decimal point are valid text characters, numbers can be interpreted as strings.

Example

```
10  INPUT "Please type a number"; A$
    PRINT A$
```

Output

```
Please type a number? 25.5
25.5
```

BASIC–PLUS–2 interprets the input as a four-character string instead of a numeric value.

You can type strings with or without quotation marks. However, if you want to input a string containing a comma, you should enclose the string in quotation marks or use the INPUT LINE or LINPUT statement. If you use the INPUT statement and do not enclose the string and comma in quotation marks, BASIC–PLUS–2 treats the comma as a delimiter and assigns only part of the string to the variable.

When using quotation marks, be sure to specify an end quotation mark as well as a beginning quotation mark. If you leave out the end quotation mark, BASIC–PLUS–2 signals the error "Data format error" (ERR=50).

5.1.1.2 The INPUT LINE and LINPUT Statements

The INPUT LINE and LINPUT statements prompt you for string data while the program executes.

The INPUT LINE statement accepts and stores all characters, including quotation marks, semicolons, and commas, up to and including the line terminator or terminators. LINPUT accepts all characters up to, but not including, the line terminator or terminators.

In the following example, both INPUT LINE and LINPUT treat your input as a string literal. Therefore, BASIC–PLUS–2 interprets quotation marks as characters, not as string delimiters. Note that the INPUT LINE statement includes the line terminator as a part of the string literal. The carriage return and line feed combination causes double spacing in the printed output.

Example

```
10  INPUT LINE A$
    LINPUT B$
    LINPUT C$
    PRINT A$, B$, C$
    PRINT "DONE"
    END
```

Output

```
? "NOW, LOOK HERE!", HE SAID.
? "NOT THERE, HERE!"

"NOW, LOOK HERE!", HE SAID.

"NOT THERE, HERE!"
DONE
```

The INPUT, INPUT LINE, and LINPUT statements can accept data from a terminal or a terminal-format file. See Section 5.3 for information on I/O to terminal-format files.

5.1.1.3 Enabling and Disabling the Question Mark Prompt

With the SET PROMPT statement, BASIC–PLUS–2 allows you to enable and disable the question mark prompt.

By default, BASIC–PLUS–2 displays the question mark prompt.

Example

```
110 INPUT "Please input 3 integer values";A%, B%, C%
```

Output

```
Please input 3 integer values?
```

To disable the question mark prompt, specify the SET NO PROMPT statement, as shown in the following example.

Example

```
100  SET NO PROMPT
110  INPUT "Please input 3 integer values";A%, B%, C%
```

Output

```
Please input 3 integer values
```

When you disable the question mark prompt, you can specify your own prompt at the end of each prompt string. The following example inserts a colon at the end of the prompt string.

Example

```
100  SET NO PROMPT
110  INPUT "Please enter your name: ";Employee_name$
```

Output

```
Please enter your name:
```

Now, if you specify the SET PROMPT statement, BASIC–PLUS–2 displays both the colon and a question mark.

Example

```
SET PROMPT
INPUT "Please enter your name: ";Employee_name$
```

Output

```
Please enter your name: ?
```

The SET [NO] PROMPT statement is valid for INPUT, LINPUT, INPUT LINE, and MAT INPUT statements. If the prompt is disabled, any one of the following commands enables it:

- SET PROMPT statement
- CHAIN statement
- NEW, OLD, RUN, or SCRATCH command

5.1.2 Providing Input from the Source Program

The following sections describe the BASIC–PLUS–2 statements READ, DATA, and RESTORE. In order to use READ and DATA statements, you must know what data is required while writing the program. These statements do not stop to request data while the program executes. Therefore, your program executes faster than it would if it contained INPUT statements.

The RESTORE statement lets you use the same data items more than once.

5.1.2.1 The READ and DATA Statements

The READ statement reads values from a data block. A data pointer keeps track of the data read. Each time the READ statement requests data, BASIC–PLUS–2 retrieves the next available constant from a DATA statement. The DATA statement contains the values that the READ statement reads. In a DATA statement, integer constants must be whole numbers; they cannot be followed by a percent sign (%). The following program example causes an error because the integer constants in the DATA statement contain percent signs.

Example

```
10   ON ERROR GOTO 400
20   DATA 1%, 2%, 3%
30   READ A%, B%, C%
40   PRINT A% + B% + C%
50   GOTO 500
400  PRINT "ERROR NUMBER IS "; ERR
     PRINT "ERROR AT LINE "; ERL
     PRINT "ERROR MESSAGE IS "; ERT$(ERR)
     RESUME 500
500  END
```

Output

```
ERROR NUMBER IS 50
ERROR AT LINE 100
ERROR MESSAGE IS %Data format error
```

A READ statement is not valid without at least one DATA statement. If your program contains a READ statement but no DATA statement, BASIC–PLUS–2 signals the compile-time error "READ without DATA."

READ statements can appear either before or after their corresponding DATA statements. The only restriction is that the DATA statements must be in the same order as their corresponding READ statements.

You can have more than one DATA statement in a program. DATA statements are ignored without at least one READ statement. As seen in the following example, you can use an ampersand (&) to continue a DATA statement.

Example

```
100 DATA "ABRAMS", BAKER, CHRISTENSON, &
        DOBSON, "EISENSTADT", FOLEY
```

Note that comment fields are not allowed in DATA statements. For example, the following statements cause *A$* to contain the string ABC!COMMENT.

Example

```
100  READ A$
     DATA ABC          !COMMENT
```

When you compile a program, BASIC–PLUS–2 creates one data block for each program module. A data block contains the values in all DATA statements in that program module. These values are stored in line number order. Each time BASIC–PLUS–2 executes a READ statement, it retrieves the next value in the data block. DATA statements cannot be shared between program modules.

If you assign alphabetic characters to a numeric variable, BASIC–PLUS–2 signals the error message "Data format error" (ERR=50). If you specify more variables in a READ statement than there are values in the DATA statement, BASIC–PLUS–2 signals the error message "Out of data" (ERR=57). If a DATA statement contains more values than variables specified in the READ statement, BASIC–PLUS–2 ignores the excess data and does not signal an error.

The following example of READ and DATA mixes string and floating-point data types. The first READ statement reads the first data item in the program: "The diameter is." The second READ statement reads the second data item: 40.5.

Example

```
100  DATA "The diameter is"
     DATA 40.5
     READ text$
     READ radius
     DIAMETER = radius * 2
     PRINT text$; DIAMETER
     END
```

Output

```
The diameter is 81
```

5.1.2.2 The RESTORE Statement

The RESTORE statement allows you to read the same data more than once. It has no effect without READ and DATA statements.

RESTORE resets the data pointer to the beginning of the first DATA statement in the program unit. You can then read data values again. Consider the following program.

Example

```
10 READ B,C,D
20 RESTORE
30 READ E,F,G
40 DATA 6,3,4,7,9,2
50 END
```

The READ statement in line 10 reads the first three values in the DATA statement:

B=6
C=3
D=4

The RESTORE statement resets the pointer to the beginning of line 40. During the second READ statement (line 30), the first three values are read again:

E=6
F=3
G=4

Without the RESTORE statement, line 30 would assign the following values:

E=7
F=9
G=2

5.2 Program Output

The PRINT statement displays data on your terminal during program execution. BASIC–PLUS–2 evaluates expressions before displaying results. Note that besides the PRINT statement, you can also use the PRINT USING statement to print and format data. For information about the PRINT USING statement, see Chapter 13.

When you use the PRINT statement, BASIC–PLUS–2 does the following:

- Precedes positive numbers with a space and negative numbers with a minus sign (–)
- Prints a space after each number
- Prints strings without leading or trailing spaces

When an element in a list is not a simple variable or constant, BASIC–PLUS–2 evaluates the expression before printing the value.

Example

```
10  A = 45
    B = 55
    PRINT A + B
    END
```

Output

```
100
```

However, BASIC–PLUS–2 interprets text inside quotation marks as a string literal.

Example

```
10  A = 45
    B = 55
    PRINT "A + B"
    END
```

Output

```
A + B
```

The PRINT statement without an expression prints a blank line.

Example

```
10  PRINT "This example leaves a blank line"
    PRINT
    PRINT "between two lines."
    END
```

Output

```
This example leaves a blank line

between two lines.
```

5.2.1 Print Zones—the Comma and the Semicolon

A terminal line contains zones that are 14 character positions wide. The number of zones in a line depends on the width of your terminal. A 72-character line contains 5 zones, which start in columns 1, 15, 29, 43, and 57. A 132-character line has additional print zones starting at columns 71, 85, 99, and 113.

The PRINT statement formats program output into these zones in different ways, depending on the character that separates the elements to be printed. If a comma precedes the PRINT item, BASIC–PLUS–2 prints the item at the beginning of the next print zone. If the last print zone on a line is filled, BASIC–PLUS–2 continues output at the first print zone on the next line.

Example

```
100  INPUT A ,B ,C ,D ,E ,F
     PRINT A ,B ,C ,D ,E ,F
     END
```

Output

```
? 5,10,15,20,25,30
 5             10             15             20             25
 30
```

BASIC–PLUS–2 skips one print zone for each extra comma between list elements. For example, the following program prints the value of *A* in the first zone and the value of *B* in the third zone.

Example

```
10  A = 5
    B = 10
    PRINT "first zone",,"third zone"
    PRINT A,,B
    END
```

Output

```
first zone                    third zone
 5                             10
```

If you separate print elements with a semicolon, BASIC–PLUS–2 does not move to the next print zone. In the following example, the first PRINT statement prints two numbers. (Printed numbers are preceded by a space or a minus sign and followed by one space.) The second PRINT statement prints two strings.

Example

```
10  PRINT 10; 20
    PRINT "ABC"; "XYZ"
    END
```

Output

```
 10  20
ABCXYZ
```

Whether you use a comma or a semicolon at the end of the PRINT statement, the cursor remains at its current position until BASIC–PLUS–2 encounters another PRINT or INPUT statement. In the following example, BASIC–PLUS–2 prints the current values of *X*, *Y*, and *Z* on one line because a comma follows the last item in the line PRINT X, Y.

Example

```
100  INPUT X,Y,Z
     PRINT X,Y,
     PRINT Z
     END
```

Output

```
? 5,10,15
 5             10                   15
```

The following example shows PRINT statements using a comma, a semicolon, and no formatting character after the last print item.

Example

```
     !No comma after I%, so each element
     !Prints on its own line
     !
100  PRINT I% FOR I% = 1% TO 10%
     PRINT
     !
     !A comma follows J%, so each
     !element prints in a separate zone
     !
200  PRINT J%, FOR J% = 1% TO 10%
210  PRINT
     !
     !A semicolon follows K%, so print
     !elements are packed together
     !
210  PRINT K%; FOR K% = 1% TO 10%
300  END
```

Output

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

1           2              3              4               5
6           7              8              9               10

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

As seen in the following example, commas and semicolons also let you control the placement of string output.

Example

```
200  PRINT "first zone",,"third zone",,"fifth zone"
300  END
```

Output

```
first zone                  third zone                  fifth zone
```

The extra comma between strings causes BASIC–PLUS–2 to skip another print zone. In the following example, the first string is longer than the print zone. When the two strings are printed, the second string begins in the third print zone because that is the next available print zone after the first string is printed.

Example

```
200  PRINT "abcdefghijklmnopqrstuvwxyz","foo"
210  PRINT "first zone","second zone","third zone"
```

Output

```
abcdefghijklmnopqrstuvwxyz  foo
first zone    second zone   third zone
```

5.2.2 Output Format for Numbers and Strings

BASIC–PLUS–2 prints strings exactly as you enter them, with no leading or trailing spaces. It does not print quotation marks unless they are delimited by another matching pair.

Example

```
100  PRINT 'PRINTING "QUOTATION" MARKS'
120  END
```

Output

```
PRINTING "QUOTATION" MARKS
```

BASIC–PLUS–2 follows these rules for printing numbers:

- When you print numeric fields, BASIC–PLUS–2 precedes each number with a space or a minus sign and follows it with a space.
- BASIC–PLUS–2 does not print trailing zeros to the right of the decimal point. If all digits to the right of the decimal point are zeros, BASIC–PLUS–2 omits the decimal point as well.
- When you print LONG integers, BASIC–PLUS–2 prints up to 10 significant digits.

BASIC–PLUS–2 follows these rules for printing floating-point numbers:

- If a floating-point number can be represented exactly by six decimal digits (or fewer) and, optionally, a decimal point, BASIC–PLUS–2 prints it that way.
- When you print a floating-point number whose integer portion is six decimal digits or less (for example, 1234.567), BASIC–PLUS–2 rounds the number to six digits (1234.57). If the integer portion of the number is seven decimal digits or larger, BASIC–PLUS–2 rounds the number to six digits and prints it in E format. See the *BASIC–PLUS–2 Reference Manual* for more information about E format.
- When you print a floating-point number with magnitude between 0.1 and 1, BASIC–PLUS–2 rounds it to six digits. When you print a floating-point number with more than six digits, and with magnitude smaller than 0.1, BASIC–PLUS–2 rounds it to six digits and prints it in E format.

The PRINT statement displays only up to six digits of precision for floating-point numbers. This corresponds to the precision of the SINGLE data type. To display the extra digits in DOUBLE numbers, you must use the PRINT USING statement. See Chapter 13 for more information on the PRINT USING statement.

The following example shows how BASIC–PLUS–2 prints various numbers with single precision:

Example

```
100  FOR I = 1 TO 20
        PRINT 2^(-I),I,2^I
120  NEXT I
140  END
```

Output

```
 .5              1              2
 .25             2              4
 .125            3              8
 .0625           4              16
 .03125          5              32
 .015625         6              64
 .78125E-02      7              128
 .390625E-02     8              256
 .195313E-02     9              512
 .976563E-03     10             1024
 .488281E-03     11             2048
 .244141E-03     12             4096
 .12207E-03      13             8192
 .610352E-04     14             16384
 .305176E-04     15             32768
```

```
.152588E-04     16          65536
.767939E-05     17          131072
.38147E-05      18          262144
.190735E-05     19          524288
.953674E-06     20          .104858E+07
```

5.3 Terminal-Format Files

Terminal-format files let you perform simple I/O to disk files. The records in a terminal-format file must be accessed sequentially. That is, you must access the records in the file one by one, from the first to the last. You can add new records only at the end of the file.

Just as the INPUT, LINPUT, and INPUT LINE statements receive information from a terminal, the INPUT #, LINPUT #, and INPUT LINE # statements receive information from a terminal-format file. And, as the PRINT statement sends information to the terminal, the PRINT # statement sends information to a terminal-format file.

Terminal-format files are very useful for creating files to be printed on a line printer, or for supplying a program with moderate amounts of input. However, if you want to use the same file for both input and output, you should not use terminal-format files. Instead, use sequential, relative, or indexed files. For more information, see Chapter 12.

Note that you do not have to use a program to create a terminal-format file. You can use a text editor to create a file and insert data, then use a BASIC–PLUS–2 program to open the file and retrieve the data.

5.3.1 Opening and Closing a Terminal-Format File

You use the OPEN statement to create a file, or to gain access to an existing file. If you do not specify either FOR INPUT or FOR OUTPUT in the OPEN statement, BASIC–PLUS–2 tries to open an existing file. If the file does not exist, BASIC–PLUS–2 creates a new one.

The channel specification lets you associate a number with the file for as long as the file is open. All I/O operations to or from the file use this number.

When you are finished accessing a file, you close it with the CLOSE statement.

5.3.2 Writing Records to a Terminal-Format File

The following example receives information from a terminal, then writes the information to a terminal-format file as a report.

Example

```
30  PRINT "This program creates a daily sales report file named SALES.DAT"
40  OPEN "SALES.DAT" FOR OUTPUT AS FILE #4%
50  PRINT #4%, "Salesperson","Sales Area","Items Sold"
60  PRINT #4%
70  INPUT "How many salespersons for today's report"; sales_persons%
80  FOR I% = 1% TO sales_persons%
        INPUT "Salesperson's name"; s_name$
        INPUT "Sales area"; area$
        INPUT "Number of items sold"; items_sold%
        PRINT #4%, s_name$, area$, items_sold%
100 NEXT I%
110 CLOSE #4%
120 END
```

Output

```
This program creates a daily sales report file named SALES.DAT
How many salespersons for today's report? 3
Salesperson's name? JONES
Sales area? NJ
Items sold? 5
Salesperson's name? SMITH
Sales area? NH
Items sold? 6

Salesperson's name? BAINES
Sales area? VT
Items sold? 8
```

This program first prints a header explaining its purpose, then opens a terminal-format file on channel #4. After this file is opened, the two PRINT # statements place an explanatory header followed by a blank line into the file.

The program then prompts you for the number of salespersons for which data is to be entered. The FOR...NEXT loop prompts for the name, sales area, and items sold for each salesperson. Note that the FOR...NEXT loop executes only as many times as there are salespersons. See Chapter 6 for more information about FOR...NEXT loops.

After the data has been entered for each salesperson, the program writes this information to the terminal-format file. Because the response to the first question was 3, the FOR...NEXT loop executes three times.

After the last item has been printed to the file, the program closes the file and ends. When you display the file with the DCL command TYPE, you see that the information is printed under the proper headers. You can also print the file on a line printer. Note that the PRINT # statement formats the output in print zones as the PRINT statement does.

Example

```
$ TYPE SALES.DAT

Salesman      Sales Area    Items Sold

JONES         NJ             5
SMITH         NH             6
BAINES        VT             8
```

6

Control Statements

BASIC–PLUS-2 normally executes statements sequentially. *Control statements* let you change this sequence of execution. BASIC–PLUS-2 control statements can alter the sequence of program execution at several levels:

- Statement modifiers control the execution of a single statement.
- Loops or decision blocks control the execution of a block of statements.
- Branching statements such as GOTO and ON GOTO pass control to statements or local subroutines.
- The EXIT and ITERATE statements explicitly control loops or decision blocks.
- The SLEEP, WAIT, STOP and END control statements suspend or halt the execution of your entire program.

This chapter describes all of the BASIC–PLUS-2 control statements.

6.1 Statement Modifiers

Statement modifiers are control structures that operate on a single statement. Statement modifiers let you execute a statement conditionally or create an implied loop. BASIC–PLUS-2 has five statement modifiers:

- IF
- UNLESS
- FOR
- UNTIL
- WHILE

A statement modifier affects only the statement immediately preceding it. You can modify only executable statements; declarative statements are not modifiable.

6.1.1 The IF Modifier

The IF modifier tests a conditional expression. If the conditional expression is true, BASIC–PLUS-2 executes the statement. If it is false, BASIC–PLUS-2 does not execute the modified statement but continues execution at the next program statement. The following is an example of a statement using the IF modifier:

```
10   PRINT A IF (A < 5)
```

6.1.2 The UNLESS Modifier

The UNLESS modifier tests a conditional expression. BASIC–PLUS-2 executes the modified statement only if the conditional expression is false. Like the IF modifier, the UNLESS modifier operates on a single statement:

```
20   PRINT A UNLESS (A < 5)
```

This is equivalent to:

```
30    PRINT A IF A >= 5
```

6.1.3 The FOR Modifier

The FOR modifier creates a loop on a single line. The following is an example of an implied loop created by a FOR modifier:

```
20    A = A + 1 FOR I% = 1% TO 10%
```

6.1.4 The UNTIL Modifier

The UNTIL modifier, like the FOR modifier, creates a single-line loop. However, instead of using a formal loop variable, you specify the terminating condition with a conditional expression. The modified statement executes repeatedly as long as the condition is false. For example:

```
40    B = B + 1 UNTIL (A  - B) < 0.0001
```

Because of precision limitations, you should not use real number calculations in UNTIL loops. For example:

```
10    Z = Z + 1 UNTIL Z/5 = 100
```

Because *Z*/5 may never exactly equal 100, the loop could execute indefinitely.

6.1.5 The WHILE Modifier

The WHILE modifier repeats a statement as long as a conditional expression is true. Like the UNTIL and FOR modifiers, it lets you create single-line loops. In the following example, BASIC–PLUS-2 replaces the value of *A* with *A*/2 as long as the absolute value of *A* is greater than 1/10. Note that you can inadvertently create an infinite loop if the terminating condition is never reached.

```
20    A = A / 2 WHILE ABS(A) > 0.1
```

6.1.6 Nested Modifiers

You can append more than one modifier to a statement. This is called *nesting* modifiers. BASIC–PLUS-2 evaluates nested modifiers from right to left. If the test of the rightmost modifier fails, control passes to the next statement, not to the preceding modifier on the same line.

In the following example, BASIC–PLUS-2 first tests the rightmost qualifier of the first PRINT statement. Because this condition is not met, BASIC–PLUS-2 tests the rightmost qualifier of the second PRINT statement. Once again, this condition is not met. Because both conditions are met in the third PRINT statement, BASIC–PLUS-2 prints the value of *C*.

Example

```
30    A = 5
      B = 10
      C = 15
40    PRINT "A =";A IF A = 5 UNLESS C = 15
      PRINT "B =";B UNLESS C = 15 IF B = 10
      PRINT "C =";C IF B = 10 UNLESS C = 5
50    END
```

Output

```
C = 15
```

6.2 Loops

Loops allow you to repeat the execution of a set of statements. This set of statements is called a *loop block*. There are three types of BASIC–PLUS-2 program loops:

- FOR...NEXT
- WHILE...NEXT
- UNTIL...NEXT

If you know how many times you want a loop to execute, that is, the number of *iterations*, you can use a FOR...NEXT loop. If you do not know the exact number of iterations when the loop begins execution, you can use either a WHILE...NEXT or an UNTIL...NEXT loop.

Note that all of these types of loops can be nested: that is, lexically located one inside another.

6.2.1 FOR...NEXT Loops

In a FOR...NEXT loop, you specify a loop control variable (loop index) that determines the number of loop iterations. This number must be a scalar (unsubscripted) variable. When BASIC–PLUS-2 begins execution of a FOR...NEXT loop, the starting and ending values of the loop control variable are known.

The FOR statement assigns the control variable a starting value and a terminating value. You can use the optional STEP clause to specify the amount to be added to the loop control variable after each loop iteration. For instance, the first example assigns the values 1 through 10 to consecutive array elements 1 through 10 of *New_array*, whereas the second example assigns consecutive multiples of 2 to the odd-numbered elements of *New_array*.

Example 1

```
10     FOR I% = 1% TO 10%
           New_array(I%) = I%
       NEXT I%
```

Example 2

```
20     FOR I% = 1% TO 10% STEP 2
           New_array(I%) = I% + 1%
       NEXT I%
```

Note that the starting, ending, and step values can be *run-time expressions*. You can have BASIC–PLUS-2 calculate these values when the program runs, as opposed to using a constant value.

The following example assigns sales information to array *Sales_data*. The number of iterations depends on the value of the variable *Days_in_month*, which represents the number of days in that particular month.

Example

```
30     FOR I% = 1% TO Days_in_month
           Sales_data(I%) = Quantity_sold
       NEXT I%
```

When a FOR loop block executes, the BASIC–PLUS-2 compiler:

1. Evaluates the starting value and assigns it to the control variable.

2. Evaluates the ending value and the step value and assigns these results to temporary storage locations.
3. Tests whether the ending value has been exceeded. If the ending value has already been exceeded, BASIC–PLUS-2 executes the statement following the NEXT statement. If the ending value has not been exceeded, BASIC–PLUS-2 executes the statements in the loop.
4. Adds the step value to the control variable and transfers control to the FOR statement, which tests whether the ending value has been exceeded.

Step 3 and step 4 are repeated until the ending value is exceeded.

Note that BASIC–PLUS-2 performs the test before the loop executes. When the control variable exceeds the ending value, BASIC–PLUS-2 exits the loop, and then subtracts the step value from the control variable. This means that after loop execution, the value of the control variable is the value last used in the loop, not the value that caused loop termination. If the starting value is greater than the ending value, and the step value is positive, the loop will not execute.

Because the starting, ending, and step values can be numeric expressions, they are not evaluated until the program runs. This means that you can have a FOR...NEXT loop that does not execute. The following example prompts the user for the starting, ending, and step values for a loop, and then tries to execute that loop. The loop executes zero times because it is impossible to go from 0 to 5 using a step value of –1.

Example

```
10    counter% = 0%

20    INPUT "Start"; start%
      INPUT "Finish"; finish%
      INPUT "Step value"; step_val%

30    FOR I% = start% TO finish% STEP step_val%
          counter% = counter% + 1%
      NEXT I%

40    PRINT "This loop executed"; counter%; "times."
```

Output

```
Start? 0
Finish? 5
Step value? -1
This loop executed 0 times.
```

Whenever possible, you should use integer variables to control the execution of FOR...NEXT loops because some decimal fractions cannot be represented exactly in a binary computer, and the calculation of floating-point control variables is subject to this inherent imprecision.

In the following example, the first loop uses an integer control variable while the second uses a floating-point control variable. The first loop executes 100 times and the second 99 times. After the ninety-ninth iteration of the second loop, the internal representation of the value of *Floating_point_variable* exceeds 10 and BASIC–PLUS-2 exits the loop. Because the first loop uses integer values to control execution, BASIC–PLUS-2 does not exit the loop until *Integer_variable* equals 100.

Example

```
10      Loop_count_1 = 0%
        Loop_count_2 = 0%

20      FOR Integer_variable = 1% to 100% STEP 1%
               Loop_count_1 = Loop_count_1 + 1%
        NEXT Integer_variable
30      FOR Floating_point_variable = 0.1 to 10 STEP 0.1
                Loop_count_2 = Loop_count_2 + 1%
        NEXT Floating_point_variable
40      PRINT "Integer loop count:"; Loop_count_1
        PRINT "Integer loop end  :"; Integer_variable
        PRINT "Real loop count:   "; Loop_count_2
        PRINT "Real loop end:     "; Floating_point_variable
```

Output

```
Integer loop count:  100
Integer loop end:    100
Real loop count:     99
Real loop end:       9.9
```

If you need to use floating-point values in a loop, you should initialize a floating-point variable and increment it within the loop.

Example

```
20      Real_counter = 0.1
        Count_loop:
30      FOR Integer_variable = 1% TO 100000%
           Real_counter = Real_counter + .1
        NEXT Integer_variable
```

Although it is not recommended programming practice, you can assign a value to a FOR...NEXT loop's *control variable* while in the loop. This affects the number of times a loop executes. For example, assigning a value that exceeds the ending value of a loop will cause the loop's execution to end as soon as BASIC–PLUS-2 performs the termination test in the FOR statement.

Assigning values to ending or step variables, however, has no effect at all on the loop's execution.

6.2.2 WHILE...NEXT Loops

A WHILE...NEXT statement uses a conditional expression to control loop execution; the loop is executed as long as a given condition is true. A WHILE...NEXT loop is useful when you do not know how many loop iterations are required.

In the following example, the first statement asks you to input data and then enter DONE when you are finished. After you enter the first piece of input, BASIC–PLUS-2 executes the WHILE...NEXT loop. If the first input value is not "DONE," the loop executes and prompts you for another input value. Once you enter the input value, the WHILE...NEXT loop once again checks to see if this value corresponds to "DONE." The loop continues executing until you enter "DONE" in response to the prompt.

Example

```
10    INPUT 'Type "DONE" when finished'; Answer
20    WHILE (Answer <> "DONE")
   .
   .
   .
         INPUT "More data"; Answer
      NEXT
```

Note that the NEXT statement in the WHILE...NEXT and UNTIL...NEXT loops does not increment a control variable; your program must change a variable in the conditional expression or the loop will execute indefinitely.

The evaluation of the conditional expression determines whether the loop executes. The test is performed (that is, the conditional expression is evaluated) before the first iteration; if the value is false (0), the loop does not execute.

It can be useful to intentionally create an infinite loop by coding a WHILE...NEXT loop whose conditional expression is always true. Of course, when doing this, you must still take care to provide a way out of the loop. You can do this with an EXIT statement or by trapping a run-time error. See Chapter 15 for more information about trapping run-time errors.

6.2.3 UNTIL...NEXT Loops

An UNTIL...NEXT loop executes repeatedly for as long as the conditional expression is false. Note that in UNTIL...NEXT and WHILE...NEXT loops, the NEXT statement does not increment a control variable. You must explicitly change a variable in the conditional expression or the loop will execute indefinitely.

It is possible to code the example in Section 6.2.2 as an UNTIL...NEXT loop as shown in the following example. These loops are equivalent except for the logical sense of the termination test (WHILE *Answer* <> "DONE" as opposed to UNTIL *Answer* = "DONE").

Example

```
10    INPUT 'Type "DONE" when finished.'; Answer

20    UNTIL (Answer = "DONE")
  .
  .
  .
         INPUT "More data"; Answer
      NEXT
```

UNTIL and FOR loops differ because BASIC–PLUS-2 exits UNTIL loops as soon as the test for the terminating condition is met. This test occurs after BASIC–PLUS-2 executes the NEXT statement and before it executes the UNTIL statement. For example, the following loop executes 10 times. When BASIC–PLUS-2 exits the FOR loop, *J%* equals 10.

Example

```
10    FOR J% = 1% to 10%
          A = A + 1
          PRINT A
      NEXT J%
```

The following UNTIL loop executes only nine times. After the ninth iteration, the conditional expression is true; control then passes out of the loop.

Example

```
10    J% = 1%
20    UNTIL J% = 10%
         PRINT J%
         J% = J% + 1%
      NEXT
```

6.2.4 Nested Loops

When a loop block is entirely contained in another loop block, it is called a *nested* loop.

The following example declares a two-dimensional array and uses nested FOR...NEXT loops to fill the array elements with sales information. The inner loop executes 16 times for each iteration of the outer loop. This example assigns zero to each of the 256 elements of the array.

Example

```
10    DECLARE INTEGER           &
         Column_number,         &
         Row_number

      DECLARE REAL              &

         Sales_info,            &
         Two_dim_array (15%, 15%)
20     FOR Row_number = 0% TO 15%
           FOR Column_number = 0% to 15%
               INPUT "Please enter the sales information";Sales_info
               Two_dim_array (Row_number, Column_number) = Sales_info
           NEXT Column_number
       NEXT Row_number
```

Note that in nested loops the inner loop is entirely contained in the outer loop: nested loops cannot overlap.

6.3 Unconditional Branching

The GOTO statement specifies which program line the BASIC–PLUS-2 compiler is to execute next, regardless of that line's position in the program. If the statement at the target line number or label is nonexecutable (such as a REM statement), BASIC–PLUS-2 transfers control to the next executable statement following the target line number.

You can use a GOTO statement to exit from a loop; however, it is better programming practice to use the EXIT statement.

6.4 Conditional Branching

Conditional branching is the transfer of program control only when specified conditions are met. There are three BASIC–PLUS-2 statements that let you conditionally transfer control to a target statement in your program:

- The ON...GOTO...OTHERWISE statement
- The IF...THEN...ELSE statement

- The SELECT...CASE statement

6.4.1 The ON...GOTO...OTHERWISE Statement

In the ON...GOTO...OTHERWISE statement, BASIC–PLUS-2 tests the value specified after the ON keyword. If the value is 1, BASIC–PLUS-2 transfers control to the first target in the list; if the value is 2, control passes to the second target, and so on. If the value is less than 1, or greater than the number of targets in the list, BASIC–PLUS-2 transfers control to the target specified in the OTHERWISE clause.

Example

```
10     Menu:
          PRINT "Would you like to change:"
          PRINT "1.  First name"
          PRINT "2.  Last name"

20     INPUT CHOICE%
       ON CHOICE% GOTO First_name, Last_name OTHERWISE Other_choice

30     First_name:
           INPUT "First name"; firstname$
           GOTO Done
40     Last_name:
          INPUT "Last name"; lastname$
          GOTO Done
50     Other_choice:
          PRINT "Invalid choice"
          PRINT "Let's try again"
          GOTO Menu

60     Done:
          END
```

Note that if you do not supply an OTHERWISE clause and the control variable is less than 1 or greater than the number of targets, BASIC–PLUS-2 signals “ON statement out of range” (ERR = 58).

6.4.2 The IF...THEN...ELSE Statement

The IF...THEN...ELSE statement evaluates a conditional expression and uses the result to determine which block of statements to execute next. If the conditional expression is true, BASIC–PLUS–2 executes the statements in the THEN clause. If the conditional expression is false, BASIC–PLUS–2 executes the statements in the ELSE clause, if one is present. If the conditional expression is false and there is no ELSE clause, BASIC–PLUS–2 executes the statement immediately following the END IF statement.

In the following example, BASIC–PLUS–2 evaluates the conditional expression *number* < 0. If the input value of *number* is less than zero, the conditional expression is true. BASIC–PLUS–2 then executes the four statements in the THEN clause and skips the statement in the ELSE clause. BASIC–PLUS–2 transfers control to the statement following the END IF. If the value of *number* is greater than or equal to zero, the conditional expression is false. BASIC–PLUS–2 then skips the statements in the THEN clause and executes the statement in the ELSE clause.

Example

```
10    INPUT "Input number"; number

20    IF (number < 0)
        THEN
           number = number * (-1)
           PRINT "That square root is imaginary"
           PRINT "The square root of its absolute value is";
           PRINT SQR(number)
        ELSE
           PRINT "The square root is"; SQR(number)
      END IF
30    END
```

Output

```
Input number? -9
That square root is imaginary
The square root of its absolute value is 3
```

One of the most common programming errors is neglecting to terminate an IF...THEN...ELSE statement. After an IF block is executed, control is transferred to the statement immediately following the END IF. If there is no END IF, BASIC–PLUS–2 transfers control to the next line number. When this happens, any code between the keyword ELSE and the next line number becomes part of the ELSE clause. If there are no line numbers, the BASIC–PLUS–2 compiler ignores the remaining program code from the keyword ELSE to the end of the program. Therefore it is very important that you always use END IF to terminate IF statements.

In the following example, the first IF...THEN...ELSE statement is terminated by END IF, and therefore works as expected. Because the second IF...THEN...ELSE statement is not terminated by END IF, the BASIC–PLUS–2 compiler assumes that the last PRINT statement in the program is part of the second ELSE clause. When you run this program, the first IF...THEN...ELSE statement will always execute correctly. However, the final PRINT statement will execute only when the value of *On_off_val* is 1, because the compiler considers this PRINT statement to be part of the second ELSE clause.

Example

```
10  DECLARE INTEGER light_bulb
    DECLARE INTEGER circuit_switch
    DECLARE INTEGER CONSTANT Opened  = 0
    DECLARE INTEGER CONSTANT Closed =  1

    PRINT "Please enter zero or one, corresponding to the circuit"
    PRINT "Switch being open or closed"
    INPUT On_off_val
    IF On_off_val = Opened
      THEN
        PRINT "The light bulb is off."
      ELSE
        PRINT "The light bulb is on."
    END IF
    IF On_off_val = Closed
      THEN
        PRINT "The light bulb is on."
      ELSE
        PRINT "The light bulb is off."
        PRINT "That's all for now."
20  END
```

Output 1

```
Please enter zero or one, corresponding to the circuit
Switch being open or closed
? 0
The light bulb is off.
The light bulb is off.
That's all for now.
```

Output 2

```
Please enter zero or one, corresponding to the circuit
Switch being open or closed
? 1
The light bulb is on.
The light bulb is on.
```

Note that a statement in a THEN or ELSE clause can be followed by a modifier. In the following example, the modifying IF applies only to the preceding statement.

Example

```
10    IF A = B
        THEN
           PRINT A IF A = 3
        ELSE
           PRINT B IF B > 0
      END IF
```

6.4.3 The SELECT...CASE Statement

The SELECT...CASE statement lets you specify an expression (the SELECT expression), any number of possible values (cases) for the SELECT expression, and a list of statements (a case block) for each case. The select-item can be a numeric or string value. Case-items can be single or multiple values, one or more ranges of values, or relationships. When a match is found between the select-item and a case-item, the statements in the following CASE block are executed. Control is then transferred to the statement following the END SELECT statement.

In the following example, the case-item values appear to overlap; that is, the case-item that tests for values greater than or equal to 0.5 also includes the values greater than or equal to 1.0. However, BASIC–PLUS-2 executes the statements associated with the *first* matching CASE statement and then transfers control to the statement following END SELECT. In this program, each range of values is tested *before* it overlaps in the next range. Because the compiler executes the first matching CASE statement, the overlapping values do not matter.

Example

```
10     DECLARE REAL Stock_change

20     INPUT "Please enter stock price change";Stock_change

30     SELECT Stock_change

         CASE <= 0.5
            PRINT "Don't sell yet."

         CASE <= 1.0
            PRINT "Sell today."
         CASE ELSE
            PRINT "Sell NOW!"

        END SELECT
40      END
```

Output

```
Please enter stock price change? 2.1
Sell NOW!
```

If no match is found for any of the specified cases and there is no CASE ELSE block, BASIC–PLUS–2 transfers control to the statement following END SELECT without executing any of the statements in the SELECT block.

SELECT...CASE is very powerful because it lets you use run-time expressions for both select-items and case-items. The following example uses BASIC–PLUS–2 built-in string functions to examine command input.

Example

```
10     ! This program is a skeleton command processor.
       ! It recognizes three BASIC-PLUS-2 environment commands:
       !
       !      SAVE
       !      SCRATCH
       !      OLD

20     DECLARE INTEGER CONSTANT True  = -1
       DECLARE INTEGER CONSTANT False =  0

       DECLARE STRING CONSTANT Null_input = ""   !This is the null string.

       DECLARE STRING Command

30     ! Main program logic starts here.

       Command_loop:

       WHILE True          ! This loop executes until the user enters only a
                           ! carriage return in response to the prompt.

           PRINT
           PRINT "Please enter a command (uppercase only)."
           PRINT "Type a carriage return when finished."
           INPUT Command
           PRINT

40         SELECT Command

              CASE Null_input                         ! If user enters Return,
                                                      ! exit from the loop
                GOTO Done                             ! and end the program.

         ! The next three cases use the SEG$ and LEN string functions.
         ! LEN returns the length of the typed string, and SEG$ searches
         ! the string literals ("SAVE", "SCRATCH", and "OLD") for a
         ! match up to that length.  Note that if the user types an "S",
         ! it is interpreted as a SAVE command only because SAVE is the
         ! first case tested.

              CASE SEG$( "SAVE", 1%, LEN (Command) )
                PRINT "That was a SAVE command."

              CASE SEG$( "SCRATCH", 1%, LEN (Command) )
                PRINT "That was a SCRATCH command."

              CASE SEG$( "OLD", 1%, LEN (Command) )
                PRINT "That was an OLD command."

              CASE ELSE
                PRINT "Invalid command, please try again."

           END SELECT
       NEXT
```

```
50     Done:
          END
```

6.5 The EXIT and ITERATE Statements

This section describes the EXIT and ITERATE statements and shows their use with nested control structures.

The ITERATE and EXIT statements let you explicitly control loop execution. These statements can be used to transfer control to the top or bottom of a control structure.

You can use EXIT to transfer control out of any of these structures:

- FOR...NEXT loops
- WHILE...NEXT loops
- UNTIL...NEXT loops
- IF...THEN...ELSE blocks
- SELECT...CASE blocks

You can also use EXIT to transfer control out of SUB and FUNCTION subprograms, DEF functions, and programs. In the case of control structures, EXIT passes control to the first statement following the end of the control structure.

You can use ITERATE to explicitly reexecute a FOR...NEXT, WHILE...NEXT, or UNTIL...NEXT loop. EXIT and ITERATE statements can appear only within the code blocks you wish to leave or reexecute.

Executing the ITERATE statement is equivalent to transferring control to the loop's NEXT statement. The termination test is still performed when the NEXT statement transfers control to the top of the loop. In addition, transferring control to the NEXT statement means that a FOR loop's control variable is incremented.

Supplying a label for every loop lets you state explicitly which loop to leave or reexecute. If you do not supply a label for the ITERATE statement, BASIC–PLUS–2 reexecutes the innermost active loop. For example, if an ITERATE statement (that does not specify a label) is executed in the innermost of three nested loops, only the innermost loop is reexecuted.

In contrast, labeling each loop and supplying a label argument to the ITERATE statement lets you reexecute any of the loops. A label name also helps document your code. Because you must use a label with EXIT and it is

sometimes necessary to use a label with ITERATE, you should always label the structures you want to control with these statements.

The following example shows the use of both the EXIT and ITERATE statements. This program explicitly exits the loop if you enter a carriage return in response to the prompt. If you enter a string, the program prints the length of the string and explicitly reexecutes the loop.

Example

```
10    DECLARE STRING User_string

20    Read_loop:
      WHILE 1% = 1%
          LINPUT "Please type a string"; User_string

          IF User_string == ""
             THEN
                EXIT Read_loop
             ELSE
                PRINT "Length is ";LEN(User_string)
                ITERATE Read_loop
           END IF
      NEXT
30    END
```

6.6 Executing Local Subroutines

In BASIC–PLUS-2 a subroutine is a block of code accessed by a GOSUB or ON GOSUB statement. It must be in the same program unit as the statement that calls it. The RETURN statement in the subroutine returns control to the statement immediately following the GOSUB.

The first line of a subroutine can be any valid BASIC–PLUS–2 statement, including a REM statement. You do not have to transfer control to the first line of the subroutine. Instead, you can include several entry points into the same subroutine. You can also reference subroutines by using a GOSUB or ON GOSUB statement to another subroutine.

Variables and data in a subroutine are global to the program unit in which the subroutine resides.

6.6.1 The GOSUB and RETURN Statements

The GOSUB statement unconditionally transfers control to a line in a subroutine. The last statement in a subroutine is a RETURN statement, which returns control to the first statement after the calling GOSUB. A subroutine can contain more than one RETURN statement so you can return control conditionally, depending on a specified condition.

The following example first assigns a value of 5 to the variable *A*, then transfers control to the subroutine labeled *Times_two*. This subroutine replaces the value of *A* with *A* multiplied by 2. The subroutine's RETURN statement transfers control to the first PRINT statement, which displays the changed value. The program calls the subroutine two more times, with different values for *A*. Each time, the RETURN transfers control to the statement immediately following the corresponding GOSUB.

Example

```
10     A = 5
       GOSUB Times_two
       PRINT A
20     A = 15
       GOSUB Times_two
       PRINT A
30     A = 25
       GOSUB Times_two
       PRINT A

40     GOTO Done

       Times_two:
          !This is the subroutine entry point
          A = A * 2
          RETURN

50     Done:
           END
```

Output

```
10
30
50
```

Note that BASIC–PLUS-2 signals "RETURN without GOSUB" if it encounters a RETURN statement without first having encountered a GOSUB or ON GOSUB statement.

6.6.2 The ON...GOSUB...OTHERWISE Statement

The ON...GOSUB...OTHERWISE statement transfers control to one of several target subroutines depending on the value of a numeric expression. A RETURN statement returns control to the first statement after the calling ON GOSUB. A subroutine can contain more than one RETURN statement so that you can return control conditionally, depending on a specified condition.

BASIC–PLUS–2 tests the value of the integer expression. If the value is 1, control transfers to the first line number or label in the list; if the value is 2, control passes to the second line number or label, and so on. If the control variable's value is less than 1 or greater than the number of targets in the list, BASIC–PLUS–2 transfers control to the line number of the label specified in the OTHERWISE clause. If you do not supply an OTHERWISE clause and the control variable's value is less than 1 or greater than the number of targets, BASIC–PLUS–2 signals "ON statement out of range" (ERR = 58).

Example

```
10     INPUT "Please enter first integer value"; First_value%
       INPUT "Please enter second integer value"; Second_value%

20     Choice:
          PRINT "Do you want to perform:"
          PRINT "1.  Multiplication"
          PRINT "2.  Division"
          PRINT "3.  Exponentiation"

30     INPUT Selection%

       ON Selection% GOSUB Mult, Div, Expon OTHERWISE Wrong
       GOTO Done

40     Mult:
         Result% = First_value% * Second_value%
         PRINT Result%
         RETURN

50     Div:
         Result% = First_value / Second_value%
         PRINT Result%
         RETURN

60     Expon:
         Result% = First_value% ** Second_value%
         PRINT Result%
         RETURN

70      Wrong:
         PRINT "Invalid selection"
         RETURN
```

```
80     Done:
          END
```

6.7 Suspending and Halting Program Execution

There are two BASIC–PLUS–2 statements that you can use to suspend program execution:

- SLEEP
- WAIT

These statements cause BASIC–PLUS–2 either to suspend program execution for a specified time or to wait a certain period of time for user input.

After execution of the last statement, a BASIC–PLUS–2 program automatically halts and closes all files. However, you can explicitly halt program execution by using one of the following statements:

- STOP
- END

The STOP statement does not close files. It can appear anywhere in a program. The END statement closes files and must be the last statement in a main program. For more information on the STOP and END statements, see Section 6.7.3 and Section 6.7.4.

6.7.1 The SLEEP Statement

The SLEEP statement suspends program execution for a specified number of seconds. The following program waits two minutes (120 seconds) after receiving the input string, and then prints the string.

Example

```
10     INPUT "Type a string of characters"; C$
20     SLEEP 120%
30     PRINT C$
40     END
```

The SLEEP statement is useful if you have a program that depends on another program for data. Instead of constantly checking for a condition, the SLEEP statement lets you check the condition at specified intervals.

6.7.2 The WAIT Statement

You use the WAIT statement only with terminal input statements such as INPUT, INPUT LINE, and LINPUT. For example, the following program prompts for input, then waits 30 seconds for your response. If the program does not receive input in the specified time, BASIC–PLUS-2 signals "Keyboard wait exhausted (ERR=15)" and exits the program.

Example

```
10    WAIT 30%
20    INPUT "You have 30 seconds to type your password"; PSW$
30    END
```

The WAIT statement affects all subsequent INPUT, INPUT LINE, LINPUT, MAT INPUT, and MAT LINPUT statements. To disable a previously specified WAIT statement, use WAIT 0%.

In the following example, the WAIT statement in line 10 causes the INPUT statement in line 20 to wait 30 seconds for a response. The WAIT 0% statement in line 30 disables this 30-second requirement for all subsequent INPUT statements.

Example

```
10    WAIT 30%
20    INPUT "You have 30 seconds to type your  password"; PSW$
30    WAIT 0%
40    INPUT "What directory do you want to go to"; DIR$
```

6.7.3 The STOP Statement

The STOP statement is a debugging tool that lets you check the flow of program logic. STOP suspends program execution but does not close files.

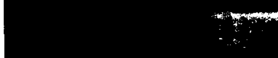
When BASIC–PLUS–2 executes a STOP statement, it prints "STOP at line lin-num."

If the program was run or compiled with the /DEBUG qualifier, the debugger number sign (#) prompt is displayed, at which time you can enter debugger commands. For example, you can enter the CONTINUE command to continue program execution or the EXIT command to return to DCL level. For a description of the BASIC–PLUS–2 debugger commands see the *BASIC–PLUS–2 Reference Manual.*

6.7.4 The END Statement

The END statement marks the end of a main program. When BASIC–PLUS–2 executes an END statement it closes all files and halts program execution. The END statement is optional. However, you should include it for good programming practice. The END statement must be the last statement in the main program.

If you run your program in the BASIC environment, the END statement returns you to BASIC–PLUS–2 command level. If you execute the program outside the BASIC environment, the END statement returns you to DCL command level.



7 Declarations and Data Types

This chapter describes how to explicitly assign data types to program variables and how to allocate and use data storage.

7.1 Declarative Statements

You use declarative statements to define objects in a BASIC–PLUS–2 program. Objects can be variables, arrays, constants, and user-defined functions within a program module. They can also be routines, variables, and constants external to the program module. Declarative statements always assign names to the objects declared and usually assign other attributes, such as a data type, to them.

You use declarative statements to assign data types to the following:

- Variables
- Arrays
- Named constants
- Values returned by functions

By declaring the objects used in your program, you make the program much easier to understand, modify, and debug.

7.2 Data Types

At its most fundamental level, a data type is a format for information storage. All information is stored in the computer as bit patterns (groups of ones and zeros). Data types specify how the computer should interpret these patterns.

BASIC–PLUS–2 programs allow three general data types: integer, floating-point, and string. Each of these general data types has unique characteristics that determine the way you use it. For example, integers are useful for numeric computations involving whole numbers, and strings provide a way to manipulate alphanumeric characters.

Within integer and floating-point data types there are further subdivisions. For example, integers can be classed as BYTE, WORD, and LONG. Choosing one of these integer subdivisions lets you control the following:

- The amount of storage required for the integer
- The range of values that the integer can accept

See Table 7–1 for more information on the range and storage requirements of these integer subtypes.

Similarly, floating-point data can be classed as SINGLE, or DOUBLE. See Table 7–1 for more information on the range and storage requirements of these floating-point subtypes. The choice you make when assigning numeric data types always involves a tradeoff between storage requirements and precision or range.

In addition to numeric and string data types, BASIC–PLUS–2 also provides a unique data type called RFA. Variables of the RFA data type require six bytes of storage and can contain only a Record File Address. RFA variables are used with RMS file I/O and the operations that can be performed on them are strictly limited. See the *BASIC–PLUS–2 Reference Manual* for more information on the RFA data type.

Traditionally, BASIC–PLUS–2 programs have had just three data types: integer, string, and floating-point. You assigned a data type to a variable by adding a suffix to the variable names; a dollar sign ($) denoted a string variable, a percent sign (%) denoted an integer variable, and variable names without suffixes denoted floating-point variables. By referencing a variable in your program, you would implicitly declare the variable with the data type indicated by the suffix character.

BASIC–PLUS–2 now lets you explicitly assign data types to variables, parameters, and functions. This feature gives you more control over the storage and precision used by your program. You can, however, still use implicit data typing in your programs. You can ensure that all program variables are explicitly declared by specifying OPTION TYPE = EXPLICIT or by using the /TYPE=EXPLICIT qualifier when you compile your programs. See Section 7.3 and the *BASIC–PLUS–2 Reference Manual* for more information on the OPTION statement.

Table 7–1 lists the keywords you use to assign data types along with their size, range, and precision.

Table 7–1 BASIC–PLUS–2 Data Types

Data Type Keyword	Size	Range	Precision (DecimalDigits)
		INTEGER	
BYTE	8 bits	–128 to +127	NA
WORD	16 bits	–32768 to +32767	NA
LONG	32 bits	–2147483648 to +2147483647	NA
		REAL	
SINGLE	32 bits	$.29 * 10^{-38}$ to $1.7 * 10^{38}$	6
DOUBLE	64 bits	$.29 * 10^{-38}$ to $1.7 * 10^{38}$	16
		STRING	
STRING	One character per byte	NA	NA
		RFA	
RFA	6 bytes	NA	NA

As shown in Table 7–1, there are four data type keywords that specify integer data. The data type INTEGER is a general data type because it specifies only that a variable contains integer data. The subtypes BYTE, WORD, and LONG specify exactly how much storage is allocated to an integer variable. If you specify the INTEGER data type, the subtype of integer variables depends on the default integer data type in effect when the program is compiled. This default is determined by the following:

- The program's OPTION statement, if present
- The qualifier (either /BYTE, /WORD, or /LONG) that you use to compile the program

Similarly, there are three data type keywords that specify floating-point data. The data type REAL is a general data type because it specifies only that a variable contains floating-point data. The subtypes SINGLE and DOUBLE specify exactly how much storage is allocated to a floating-point variable. If you specify the data type REAL, the subtype of floating-point variables depends

on the default floating-point subtype in effect when the program is compiled. This default is determined by the following:

- The OPTION statement, if present
- The /SINGLE or /DOUBLE qualifier you use to compile the program

Choosing a numeric subtype always involves a tradeoff between storage requirements and range or precision. You can reduce the size of an executable image by choosing the smallest numeric subtype that is large enough to meet your needs.

7.3 Setting the Default Data Type and Size

There are two ways to set the default data type and size for your program:

- With the OPTION statement
- With qualifiers:
  - /TYPE_DEFAULT
  - /BYTE
  - /WORD
  - /LONG
  - /DOUBLE
  - /SINGLE

The OPTION statement can override the defaults set with qualifiers. For example, the following statement sets the default integer type to be LONG.

```
10    OPTION SIZE = INTEGER LONG
```

You can have more than one OPTION statement in a program module; however, OPTION statements can be preceded only by a SUB, FUNCTION, PROGRAM, or REM statement, or by another OPTION statement.

See the *BASIC–PLUS–2 Reference Manual* for more information about the OPTION statement.

In the following example, the OPTION statement specifies the following:

- All program variables must be explicitly typed.
- All implicitly typed constants are INTEGER.
- Any variable typed as INTEGER is a LONG integer.
- Any variable typed as REAL is a DOUBLE floating-point number.

Example

```
10    OPTION  TYPE = EXPLICIT,           ! Variables must be declared        &
              CONSTANT TYPE = INTEGER,   ! All implicit constants be integers &
              SIZE = INTEGER LONG,       ! 32-bit integers by default        &
              SIZE = REAL DOUBLE         ! 64-bit floating-point
                                         ! numbers by default
```

You can create variables of other data types by explicitly declaring them with the DECLARE, COMMON, or MAP statement.

7.4 Declaring Variables Explicitly

The DECLARE statement explicitly assigns a data type or subtype to a variable, function, or constant.

The subtype you specify overrides any defaults specified in the OPTION statement, in the BASIC environment or with compilation qualifiers. For example, if you compile your program with the /WORD qualifier and then declare an integer variable to be LONG, the variable is LONG rather than WORD. Note that when you specify the data type STRING in a DECLARE statement, a dynamic string variable is used.

You can define a variable only once in a program. For example, if a variable name appears in a DECLARE statement, it cannot also appear in a COMMON or MAP statement.

You should use unique variable names to avoid confusion and make program documentation easier. For example, if you declare variable *B* to be LONG, there cannot also be a floating-point variable *B* in your program. It is possible to have both an INTEGER variable *B%* and an INTEGER variable *B* in the same program; however, this is poor programming practice.

You can also use the DECLARE statement to assign a data type and value to DEF functions and constants. See Section 7.5 for an explanation of declaring named constants.

The following statement declares the DEF function *circumference* and declares a SINGLE parameter for the function:

```
10    DECLARE WORD FUNCTION circumference(SINGLE)
```

DECLARE FUNCTION lets you assign a data type to parameters and to the value a function returns. DECLARE FUNCTION also lets you name the function without using the usual convention (beginning the function name with FN and ending the function name with a percent or dollar sign suffix). For example:

Example

```
10     DECLARE STRING FUNCTION concat (STRING, STRING) !Declare the function
   .
   .
   .
       new_string$ = concat(A$, B$)                    !Invoke the function
       DEF concat (STRING Y, STRING Z)                 !Define the function
       concat = Y + Z
       END DEF
   .
   .
   .
       END
```

This format allows only one data type in a single statement. Declaring more than one type of function requires more than one DECLARE statement.

These data typing features give you control over storage allocation. Compiling a program with OPTION TYPE = EXPLICIT is particularly useful because it causes BASIC–PLUS–2 to signal an error when an implicit variable is encountered. This prevents a typing mistake from being interpreted as a new variable. BASIC–PLUS–2 supports implicit variables for compatibility with other BASICs and also because they are useful for beginning programmers; however, it is recommended that you use explicit declarations for new program development.

7.5 Declaring Named Constants Explicitly

Constants are values that do not change during program execution. You can declare named constants within a program unit with the DECLARE statement. You can also refer to constants outside the program unit with the EXTERNAL statement.

Named constants are useful for the following reasons:

- If a commonly-used constant must be changed, you can make the change in a single place.
- They make the program easier to understand.

7.5.1 Declaring Constants Within a Program Unit

In BASIC–PLUS–2, you can specify only a single value for floating-point named constants. That is, the value assigned to named floating-point constants cannot be an expression.

The value assigned to a named constant need not be in the allowable range of the default data type; however, it must be in the valid range of the data type being declared. The following statement declares a LONG constant named *XYZ* and assigns it a value of 1000. In DECLARE CONSTANT statements, BASIC–PLUS–2 signals an overflow error only if the value is outside the range of the data type being declared.

```
10    DECLARE LONG CONSTANT XYZ = 1000
```

The following example declares a double-precision constant:

Example

```
10    DECLARE DOUBLE CONSTANT plancks = 6.6237E-27
20    INPUT "FREQUENCY"; freq
30    PRINT "ENERGY EQUALS"; plancks / freq
40    END
```

A DECLARE CONSTANT statement allows only one data type. To declare a constant of a different data type, you must use an additional DECLARE CONSTANT statement.

7.5.2 Declaring Constants External to the Program Unit

To declare constants external to the program unit, use the EXTERNAL statement. For example:

```
10    EXTERNAL WORD CONSTANT STATUS
```

The task builder automatically supplies the values for constants specified in EXTERNAL statements. For more information on using the EXTERNAL statement, see the *BASIC–PLUS–2 Reference Manual*.

7.6 Operations with Multiple Data Types

When an expression contains operands of different data types, it is called a *mixed-mode* expression. Before a mixed-mode expression can be evaluated, the operands must be converted, or *promoted* to a common data type. The result of the evaluation may also be converted depending on the data type of the variable to which it is assigned.

When evaluating mixed-mode expressions, BASIC–PLUS–2 performs promotions so that no operand loses any range or precision. When assigning values to variables, BASIC–PLUS–2 converts the result of the expression to the data type of the variable. If the value of the expression is outside the allowable range of the variable's data type, BASIC–PLUS–2 signals the error "Integer error or overflow" or "Floating-point error or overflow." Note that these overflow errors are only signaled when converting between data types; overflow during calculation is not detected.

In general, BASIC–PLUS–2 promotes operands with different data types to the lowest data type that can hold the largest and most precise possible value of either operand's data type. BASIC–PLUS–2 then performs the operation in that data type, and yields a result of that data type. If the result of the expression is assigned to a variable, BASIC–PLUS–2 converts the result to the data type of the variable. Table 7–2 lists the resulting data type for all combinations.

Table 7–2 Result Data Types in Expressions

	BYTE	WORD	LONG	SINGLE	DOUBLE
BYTE	BYTE	WORD	LONG	SINGLE	DOUBLE
WORD	WORD	WORD	LONG	SINGLE	DOUBLE
LONG	LONG	LONG	LONG	SINGLE	DOUBLE
SINGLE	SINGLE	SINGLE	SINGLE	SINGLE	DOUBLE
DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE

As Table 7–2 shows, if one operand is SINGLE and one operand is DOUBLE, BASIC–PLUS–2 promotes the SINGLE value to DOUBLE, performs the specified operation, and returns the result as a DOUBLE value. This promotion is necessary because the SINGLE data type has less precision than the DOUBLE value, whereas the DOUBLE data type can hold the largest and most precise possible SINGLE value. If BASIC–PLUS–2 did not promote the SINGLE value and the operation yielded a more precise result than was represented in SINGLE, the value would lose precision.

7.7 Allocating Static Storage

BASIC–PLUS–2 programs allocate both static and dynamic storage. The size of static storage does not change during program execution. Variables and arrays appearing in MAP or COMMON statements use static storage. Because this storage is static, all string variables appearing in MAP or COMMON statements are fixed-length strings.

Dynamic storage is allocated when the program executes. Variables and arrays declared in the following statements use dynamic storage:

- DECLARE statements
- DIMENSION statements
- Implicitly declared variables

String variables and arrays declared in this way are dynamic strings and their length can change during program execution.

MAP and COMMON statements create a named storage area called a program section (PSECT). MAP statements require a map name, but in COMMON statements the name is optional. The PSECT name is the same as the map or common name. If you do not specify a common name, BASIC–PLUS–2 supplies a default PSECT name of .$$$$. The following sections explain how to use static storage.

7.7.1 The COMMON Statement

The COMMON statement defines a named area of storage (called a PSECT). Any BASIC–PLUS–2 subprogram can access the values in a common by specifying a common with the same name. An item in a COMMON statement can be any of the following:

- A numeric variable
- A numeric array
- A fixed-length string variable
- An array of fixed-length strings
- A FILL item

The amount of storage reserved for a variable depends on its data type. You can specify a length for string variables and string array elements that appear in a COMMON statement. If you do not specify a length, the default is 16. The following statement specifies 2 bytes for *emp.code*, 3 bytes for *wage.code*, and 22 bytes for *dep.code*.

```
10    COMMON (code) STRING emp.code=2, wage.code=3, dep.code=22
```

In a single program module, multiple commons with the same name allocate storage end-to-end in a single PSECT. That is, BASIC–PLUS–2 concatenates all commons with the same name in the same program module, in the order they appear. For example, the following statements allocate storage for five LONG integers in a single PSECT named *into*.

```
10    COMMON (into) LONG call_count, sub1_count, sub2_count
      COMMON (into) LONG sub3_count, sub4_count
```

7.7.2 The MAP Statement

The MAP statement, like the COMMON statement, creates a named area of static storage. However, if a program module contains multiple maps with the same name, the maps are overlaid on the same area of storage, rather than being concatenated.

When used with the MAP clause of the OPEN statement, the storage allocated by the MAP statement becomes the record buffer for that file. Variables in the MAP statement correspond to fields in the file's records.

A map item can be one of the following:

- A numeric variable
- A numeric array
- A fixed-length string variable
- An array of fixed-length strings
- A FILL item

7.7.2.1 Single Maps

You associate a map with a record buffer by referencing the map in the OPEN statement.

The MAP statement must appear before any reference to map variables. For example, the following program uses map variables to access fields in payroll records. Changes to map variables do not change the actual records in the file. To transfer the changed variables to the file, you must use the PUT or UPDATE statement. For more information on these statements, see Chapter 12 and the *BASIC–PLUS–2 Reference Manual*.

Example

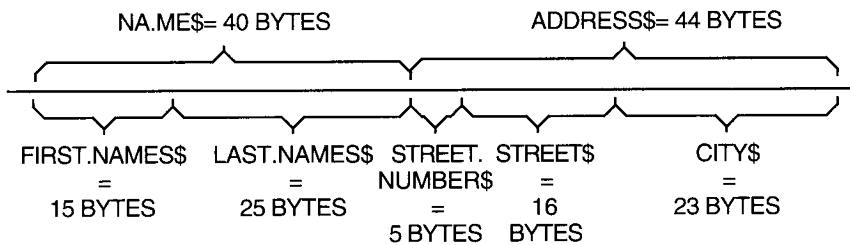
```
10      ON ERROR GOTO 50
        DECLARE INTEGER CONSTANT EOF = 11

        MAP (PAYROL) STRING emp_name, LONG wage_class,       &
                              STRING sal_rev_date, SINGLE tax_ytd

20      OPEN      "payroll.dat" FOR INPUT AS FILE #4%        &
                         ,ORGANIZATION SEQUENTIAL            &
                             ,ACCESS READ                    &
                             ,MAP PAYROL

30      OPEN "payrol.new" FOR OUTPUT AS FILE #5%             &
                      ,ORGANIZATION SEQUENTIAL               &
                      ,ACCESS WRITE                          &
                      ,MAP payrol

40      PRINT "PAYROLL VERIFICATION"

         get_loop:
          WHILE 1% = 1%
                 GET #4
                 PRINT emp_name, wage_class, sal_rev_date, tax_ytd
                 PRINT "YOU CAN CHANGE:"
                 PRINT "1. EMPLOYEE NAME"
                 PRINT "2. WAGE CLASS"
                 PRINT "3. REVIEW DATE"
                 PRINT "4. TAX YEAR-TO-DATE"
                 PRINT "5. DONE"

          read_loop:
             WHILE 1% = 1%
                 INPUT "CHANGES? ANSWER WITH YES OR NO" ; chng$
                 IF chng$ = "NO" THEN ITERATE get_loop
                         ELSE INPUT "NUMBER" ; number%
                 END IF
```

```
            SELECT number%
              CASE 1
                   INPUT "EMPLOYEE NAME"; emp_name
              CASE 2
                   INPUT "WAGE CLASS"; wage_class
              CASE 3
                   INPUT "REVIEW DATE"; sal_rev_date
              CASE 4
                   INPUT "TAX YEAR-TO-DATE"; tax_ytd
              CASE 5
                   EXIT read_loop
              CASE ELSE
                   PRINT "Invalid response -- please try again"
            END SELECT
        NEXT
     PUT #5
     NEXT
50    IF ERR = EOF
        THEN
           PRINT "End of file"
        ELSE
           ON ERROR GOTO 0
      END IF
      END
```

7.7.2.2 Multiple Maps

When a program contains more than one map with the same name, the storage allocated by the MAP statements is overlaid. This technique is useful for manipulating strings.

Figure 7–1 Multiple Maps



When you use more than one map to access a record buffer, BASIC–PLUS–2 uses the size of the largest map to determine the size of the record. However, note that the RECORDSIZE clause of the OPEN statement, if specified, will override the record size defined by a MAP statement. For more information on the RECORDSIZE clause, see Chapter 12 and the description of the OPEN statement in the *BASIC–PLUS–2 Reference Manual*.

You can also use multiple maps to interpret numeric data in more than one way. The following example creates a map area named *barray*. The first MAP statement allocates 26 bytes of storage in the form of an integer BYTE array. The second MAP statement defines this same storage as a 26-byte string named *ABC*. When the FOR...NEXT loop executes, it assigns values corresponding to the ASCII values for the uppercase letters A through Z.

Example

```
10     MAP (barray) BYTE alphabet(25)
       MAP (barray) STRING ABC = 26
       FOR I% = 0% TO 25%
          alphabet(I%) = I% + 65%
       NEXT I%
20     PRINT ABC
       END
```

Output

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

7.7.3 FILL Items

FILL items reserve space in map and common blocks and in record buffers accessed by MOVE or REMAP statements. Thus, FILL items mask parts of the record buffer and let you skip over fields and reserve space in or between data elements.

FILL formats are available for all data types. Table 7–3 summarizes the FILL formats and their default allocations if no data type is specified.

Table 7–3 FILL Item Formats, Representations, and Default Allocations

FILL Format	Representation	Bytes Used
FILL	Floating-point	4, 8, or 16
FILL(n)	n floating-point elements	4n, 8n, or 16n
FILL%	Integer (BYTE, WORD, or LONG)	1, 2, or 4

(continued on next page)

Table 7–3 (Cont.) FILL Item Formats, Representations, and Default Allocations

FILL Format	Representation	Bytes Used
FILL%(n)	n integer elements	1n, 2n, or 4n
FILL$	String	16
FILL$(n)	n string elements	16n
FILL$ = m	String	m
FILL$(n) = m	n string elements, m bytes each	m * n

Note

In the applicable formats of FILL, *n* represents a repeat count, not an array subscript. FILL(n), for example, represents *n* real elements, not *n*+1.

You can also use data type keywords with FILL and optionally data type suffixes. The data type and storage requirements are those of the last data type specified. For example:

Example

```
10    MAP (QED) STRING A, FILL$=24, LONG SSN, FILL%, REAL SAL, FILL(5)
```

In this example, the MAP statement uses data type keywords to reserve space for the following:

- A 16-character string variable *A*
- 24 bytes of padding
- LONG variable, *SSN*
- 4 bytes of padding
- REAL variable, *SAL*
- 5 floating-point numbers (which requires 20, or 40 bytes of padding, depending on the default size for floating-point numbers)

7.7.4 Using COMMON and MAP in Subprograms

The COMMON and MAP statements create a block of storage called a PSECT (program section). This common or map storage block is accessible to any subprogram. A BASIC–PLUS–2 main program and a subprogram can share such an area by referencing the same common or map name.

Example

```
10    !In a main program
      COMMON (A1) STRING A, B = 10, LONG C
   .
   .
   .
10    !In a subprogram
      COMMON (A1) STRING X, Z = 10, LONG Y
```

The previous example contains common blocks that define the following:

- A 16-character string field called *A* by the main program and *X* by the subprogram
- A 10-character string field called *B* by the main program and *Z* by the subprogram
- A 4-byte integer field called *C* by the main program and *Y* by the subprogram

Note that if a subprogram defines a common or map area with the same name as a common or map in the main program, it overlays the common or map defined in the main program.

Multiple COMMON statements with the same name behave differently depending on whether these statements are in the same program module. If they are in the same program module, then the storage for each common area is concatenated. However, if they are in different program modules, then the common areas overlay the same storage. The following COMMON statements are in the same program module; therefore, they are concatenated in a single PSECT. The PSECT contains two 32-byte strings.

```
10    COMMON (XYZ) STRING A = 32
      COMMON (XYZ) STRING B = 32
```

In contrast, the following COMMON statements are in different program modules, and thus overlay the same storage. Therefore, the PSECT contains one 32-byte string, called *A* in the main program and *B* in the subprogram.

```
10    !In the main program
      COMMON (XYZ) STRING A = 32
   .
   .
   .
20    !In the subprogram
      COMMON (XYZ) STRING B = 32
```

Although you can redefine the storage in a common section when you access it from a subprogram, you should generally not do so. Common areas should contain exactly the same variables in all program modules. To make sure of this, you should use the %INCLUDE directive, as shown in the following example:

Example

```
10     !Contents of COMMON.B2S
       COMMON (SHARE) WORD emp_num,                &

                   DOUBLE salary,                  &

                   STRING wage_class = 2
   .
   .
   .
20     !In the main program
       %INCLUDE "COMMON.B2S"
   .
   .
   .
30     !In the subprogram
       %INCLUDE "COMMON.B2S"
```

If you use the %INCLUDE directive, you can lessen the chance of a typographical error appearing in your program. For more information on using the %INCLUDE directive, see Chapter 14.

If you must redefine the variables in a PSECT, you should use the MAP statement. When you use the MAP statement, use the %INCLUDE directive to create identical maps before redefining them, as shown in the following example. The map defined in MAP.B2S is included in both program modules as a 40-byte string. This map is redefined in the subprogram, allowing the subprogram to access parts of this string.

Example

```
10     !Contents of MAP.B2S
       MAP (REDEF) STRING full_name = 40
   .
   .
   .
20     !In the main program
       %INCLUDE "MAP.B2S"
   .
   .
   .
30     !In the subprogram
       %INCLUDE "MAP.B2S"
       MAP (REDEF) STRING first_name=15, MI=1, last_name=24
```

7.8 Dynamic Mapping

Dynamic mapping lets you redefine the position of variables in a static storage area. This storage area can be either a map name or a previously declared static string variable. Dynamic mapping requires three BASIC–PLUS–2 statements:

- A declarative statement, such as a MAP statement, allocating a fixed-length storage area
- A MAP DYNAMIC statement, naming the variables whose positions can change at run time
- A REMAP statement, specifying the new positions of the variables named in the MAP DYNAMIC statement

The MAP DYNAMIC statement does not affect the amount of storage allocated. The MAP DYNAMIC statement causes BASIC–PLUS–2 to create internal pointers to the variables and array elements. Until your program executes the REMAP statement, the storage for each variable and each array element named in the MAP DYNAMIC statement starts at the beginning of the map storage area.

The MAP DYNAMIC statement is nonexecutable. With this statement, you cannot specify a string length. All string items have a length of zero until the program executes a REMAP statement.

The REMAP statement specifies the new positions of variables named in the MAP DYNAMIC statement. That is, it causes BASIC–PLUS–2 to change the internal pointers to the data. Because the REMAP statement is executable, it can redefine the pointer for a variable or array element each time the REMAP statement is executed.

With the MAP DYNAMIC statement, you can specify either a map name or a previously declared static string variable. When you specify a map name, a MAP statement with the same map name must lexically precede the MAP DYNAMIC statement.

In the following example, the MAP statement creates a storage area and names it *emp*. The MAP DYNAMIC statement specifies that the positions of variables *emp_name* and *emp_address* within the map area can be dynamically defined with the REMAP statement.

Example

```
10      DECLARE LONG CONSTANT emp_fixed_info = 4 + 9 + 2
        MAP (emp) LONG badge,                             &
                       STRING social_sec_num = 9,         &
                       BYTE name_length,                  &
                       address_length,                    &
                       FILL (60)

        MAP DYNAMIC (emp) STRING emp_name,                &
                                 emp_address

        WHILE 1%
           GET #1
           REMAP (emp) STRING FILL = emp_fixed_info,                &
                                 emp_name = name_length,            &
                                 emp_address = address_length

        NEXT
```

At the start of program execution, the storage for *badge* is the first 4 bytes of *emp*, the storage for *social_sec_num* is equal to 9 bytes and together *name_length* and *address_length* are equal to 2 bytes. The FILL keyword reserves 60 additional bytes of storage. The MAP DYNAMIC statement defines the variables *emp_name* and *emp_address* whose positions and lengths will change at run time. When executed, the REMAP statement defines the FILL area to be equal to *emp_fixed_info* and defines the positions and lengths of *emp_name* and *emp_address*.

When you specify a static string variable, it must be either a variable declared in a MAP or COMMON statement or a parameter declared in a SUB, FUNCTION, or DEF. The actual parameter passed to the procedure must be a static string variable defined in a COMMON or MAP statement.

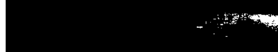
The following example shows the use of a static string variable as a parameter declared in a SUB. The MAP DYNAMIC statement specifies the input parameter, *input_rec*, as the string to be dynamically defined with the REMAP statement. In addition, the MAP DYNAMIC statement specifies a string array *A* whose elements will point to positions in *input_rec* after the REMAP statement is executed. The REMAP statement defines the length and position of each element contained in array *A*. The FOR...NEXT loop then assigns each element contained in array *A* into array *item*, the target array.

Example

```
10     SUB deblock (STRING input_rec, STRING item())

       MAP DYNAMIC (input_rec) STRING A(2)

       REMAP (input_rec) &
               A(0) = 5, &
               A(1) = 3, &
               A(2) = 4
       FOR I = 0 to 2
              item(I) = A(I)
       NEXT I
       END SUB
```

Note that dynamic map variables are local to the program module in which they reside. Therefore, REMAP only affects how that module views the buffer.

For more information on using the MAP DYNAMIC and REMAP statements, see the *BASIC–PLUS–2 Reference Manual.*



8 Functions

A *function* is a single statement or group of statements that perform operations on operands and return the result to your program. BASIC–PLUS–2 has built-in functions that perform numeric and string operations, conversions, and date and time operations. This chapter describes only a selected group of built-in functions. For a complete description of all BASIC–PLUS–2 built-in functions, see the *BASIC–PLUS–2 Reference Manual*.

This chapter also describes user-defined functions. BASIC–PLUS–2 lets you define your own functions in two ways:

- With the DEF statement
- As separately compiled subprograms (external functions)

DEF function definitions are local to a program module, while external functions can be accessed by any program module. You create local functions with the DEF statement and optionally declare them with the DECLARE statement. You create external functions with the FUNCTION statement and declare them with the EXTERNAL statement.

Once you have created and declared a function, you can invoke it just as you would a built-in function.

8.1 Built-In Functions

The functions described in this section let you perform sophisticated manipulations of string and numeric data. BASIC–PLUS–2 also provides algebraic, exponential, trigonometric, and randomizing mathematical functions. All of these functions are contained in the BASIC–PLUS–2 library of built-in functions.

8.1.1 Using Numeric Functions

Numeric functions generally return a result of the same data type as the function's parameter. For example, if you pass a DOUBLE argument to any of the trigonometric functions, they return a DOUBLE result.

If the format of a BASIC–PLUS–2 function specifies an argument of a particular data type, BASIC–PLUS–2 converts the actual argument supplied to the specified data type. For instance, if you supply an integer argument to a function that expects a floating-point number, BASIC–PLUS–2 converts the argument to floating-point. Floating-point arguments that are passed to integer functions are truncated, not rounded.

The following are some examples of BASIC–PLUS–2 built-in numeric functions.

8.1.1.1 The ABS Function

The ABS function returns a floating-point number that equals the absolute value of a specified numeric expression. The following is an example of the ABS function:

Example

```
10     READ A,B
20     DATA 10,-35.3
30     NEW_A = ABS(A)
40     PRINT NEW_A; ABS(B)
50     END
```

Output

```
10  35.3
```

ABS always returns a number of the default floating-point data type.

8.1.1.2 The INT and FIX Functions

The INT function returns the floating-point value of the largest integer less than or equal to a specified expression. INT always returns a number of the default floating-point type.

The FIX function truncates the value of a floating-point number at the decimal point. FIX always returns a number of the default floating-point type.

The following example points out the differences between the INT and FIX functions. Note that the value returned by FIX(–45.3) differs from the value returned by INT(–45.3).

Example

```
10    PRINT INT(23.553); FIX(23.553)
20    PRINT INT(3.1); FIX(3.1)
30    PRINT INT(-45.3); FIX(-45.3)
40    PRINT INT(-11); FIX(-11)
50    END
```

Output

```
 23  23
  3   3
-46 -45
-11 -11
```

8.1.1.3 The SIN, COS, and TAN Functions

The SIN, COS, and TAN functions return the sine, cosine, and tangents of an angle in radians. If you supply a floating-point argument to the SIN, COS, and TAN functions, they return a number of the same floating-point type. However, if you supply an integer argument, they convert the argument to the default floating-point data type and return a floating-point number of that type.

The following program accepts an angle in degrees, converts the angle to radians, and prints the angle's sine, cosine, and tangent:

Example

```
10    !CONVERT ANGLE (X) TO RADIANS, AND
20    !FIND SIN, COS AND TAN
30    PRINT "DEGREES", "RADIANS", "SINE", "COSINE","TANGENT"
40    FOR I% = 0% TO 5%
         READ X
         LET Y = X * 2 * PI / 360
         PRINT
         PRINT X ,Y ,SIN(Y) ,COS(Y) ,TAN(Y)
50    NEXT I%

100   DATA 0,10,20,30,360,45
200   END
```

Output

DEGREES	RADIANS	SINE	COSINE	TANGENT
0	0	0	1	0
10	.174533	.173648	.984808	.176327
20	.349066	.34202	.939693	.36397
30	.523599	.5	.866025	.57735
360	6.28319	.174846E-06	1	.174846E-06
45	.785398	.707107	.707107	1

Note

As an angle approaches 90 degrees (PI/2 radians), 270 degrees (3*PI/2 radians), 450 degrees (5*PI/2 radians) and so on, the tangent of that angle approaches infinity. If your program tries to find the tangent of such an angle, BASIC–PLUS–2 signals the error "Division by 0" (ERR=61).

8.1.1.4 The LOG10 Function

A logarithm is the exponent of some number (called a base). Common logarithms use the base 10. The common logarithm of a number *N*, for example, is the power to which 10 must be raised to equal *N*. For example, the common logarithm of 100 is 2, because 10 raised to the power 2 equals 100.

The LOG10 function returns a number's common logarithm. The following example calculates the common logarithms of all multiples of 10 from 10 to 100 inclusive:

Example

```
10    FOR I% = 10% TO 100% STEP 10%
          PRINT LOG10(I%)
20    NEXT I%
30    END
```

Output

```
1
1.30103
1.47712
1.60206
1.69897
1.77815
1.8451
1.90309
1.95424
2
```

If you supply a floating-point argument to LOG10, the function returns a floating-point number of the same data type. However, if you supply an integer argument, LOG10 converts it to the default floating-point data type and returns a value of that type.

8.1.1.5 The EXP Function

The EXP function returns the value of *e* raised to a specified power. The following example prints the value of *e* and *e* raised to the second power:

Example

```
10    READ A,B
20    DATA 1,2
30    PRINT 'e RAISED TO THE POWER'; A; " EQUALS"; EXP(A)
40    PRINT 'e RAISED TO THE POWER'; B; " EQUALS"; EXP(B)
50    END
```

Output

```
e RAISED TO THE POWER 1 EQUALS 2.71828
e RAISED TO THE POWER 2 EQUALS 7.38906
```

If you supply a floating-point argument to EXP, the function returns a floating-point number of the same data type. However, if you supply an integer argument, EXP converts it to the default floating-point data type and returns a value of that type.

8.1.1.6 The RND Function

The RND function returns a number greater than or equal to zero and less than one. The RND function always returns a floating-point number of the default floating-point data type. The RND function generates seemingly unrelated numbers. However, given the same starting conditions, a computer always gives the same results. Each time you execute a program with the RND function, you receive the same results.

Example

```
10    PRINT RND,RND,RND,RND
20    END
```

Output 1

```
.76308        .179978       .902878       .88984
```

Output 2

```
.76308        .179978       .902878       .88984
```

With the RANDOMIZE statement, you can change the RND function's starting condition and generate truly random numbers. To do this, place a RANDOMIZE statement before the line invoking the RND function. Note that the RANDOMIZE statement should be used only once in a program. With the RANDOMIZE statement, each invocation of RND returns a new and unpredictable number.

Example

```
10    RANDOMIZE
      PRINT RND,RND,RND,RND
      END
```

Output 1

```
.403732       .34971        .15302        .92462
```

Output 2

```
.404165       .272398       .261667       .10209
```

The RND function can generate a series of random numbers over any open range. To produce random numbers in the open range *A* to *B*, use the following formula:

```
200  (B-A)*RND + A
```

The following program produces 10 numbers in the open range 4 to 6:

Example

```
10    FOR I% = 1% TO 10%
         PRINT (6%-4%) * RND + 4
20    NEXT I%
30    END
```

Output

```
5.52616
4.35996
5.80576
5.77968
4.77402
4.95189
5.76439
4.37156
5.2776
4.53843
```

8.1.2 Using Data Conversion Functions

BASIC–PLUS–2 provides built-in functions that can do the following:

- Convert a 1-character string to the character's ASCII value and vice versa
- Translate strings from one data format to another, for example, EBCDIC to ASCII

The following sections describe some of these functions.

8.1.2.1 The ASCII Function

The ASCII function returns the numeric ASCII value of a string's first character. The ASCII function returns an integer value between 0 and 255, inclusive. For instance, in the following example, the PRINT statement prints the integer value 66 because this is the ASCII value equivalent of an uppercase *B*, the first character in the string.

Example

```
10    test_string$ = "BAT"
20    PRINT ASCII(test_string$)
30    END
```

Output

```
66
```

Note that the ASCII value of a null string is zero.

8.1.2.2 The CHR$ Function

The CHR$ function returns the character whose ASCII value you supply. If the ASCII integer expression that you supply is less than zero or greater than 255, BASIC–PLUS–2 treats it as a modulo 256 value. In other words, BASIC–PLUS–2 treats the integer expression as the remainder of the actual supplied integer divided by 256. Therefore, CHR$(325) is equivalent to CHR$(69) and CHR$(–1) is equivalent to CHR$(255).

The following program outputs the character whose ASCII value corresponds to the input value modulo 256:

Example

```
10    PRINT "THIS PROGRAM FINDS THE CHARACTER WHOSE"
30    PRINT "VALUE (MODULO 256) YOU TYPE"
50    INPUT value%
70    PRINT CHR$(value%)
90    END
```

Output 1

```
THIS PROGRAM FINDS THE CHARACTER WHOSE
VALUE (MODULO 256) YOU TYPE
 ? 69
E
```

Output 2

```
THIS PROGRAM FINDS THE CHARACTER WHOSE
      VALUE (MODULO 256) YOU TYPE
 ? 1093
E
```

8.1.3 Using String Numeric Functions

Numeric strings are numbers represented by ASCII characters. A numeric string consists of an optional sign, a string of digits, and an optional decimal point. You can use E notation in a numeric string for floating-point constants.

The following sections describe some of the BASIC–PLUS–2 numeric string functions.

8.1.3.1 The FORMAT$ Function

The FORMAT$ function converts a numeric value to a string. The output string is formatted according to a string you provide. The expression you give this function can be any string or numeric expression. The format string must contain at least one PRINT USING format field. The formatting rules are the same as those for printing numbers with PRINT USING. See Chapter 13 for more information on the PRINT USING statement and formatting rules.

Example

```
10    A = 5
      B$ = "##.##"
      Z$ = FORMAT$(A, B$)
30    PRINT Z$
40    END
```

Output

```
5.00
```

8.1.3.2 The NUM$ and NUM1$ Functions

The NUM$ function evaluates a numeric expression and returns a string of characters formatted as the PRINT statement would format it. The returned numeric string is preceded by one space for positive numbers and by a minus sign for negative numbers. The numeric string is always followed by a space as shown in the following example.

Example

```
10    PRINT NUM$(7465097802134)
20    PRINT NUM$(-50)
30    END
```

Output

```
.74651E+13
-50
```

The NUM1$ function translates a number into a string of numeric characters. NUM1$ does not return leading or trailing spaces or E format. The following example illustrates the use of the NUM1$ function:

Example

```
10    PRINT NUM1$(PI)
20    PRINT NUM1$("97.5"D * "30456.23"D + "30385.1"D)
30    PRINT NUM1$(1E-38)
40    END
```

Output

```
3.14159
2999870
.00000000000000000000000000000000000001
```

NUM1$ returns up to 6 digits of accuracy for single-precision real numbers, up to 16 digits of accuracy for double-precision numbers, and up to 10 digits of accuracy for LONG integers.

The following example shows the difference between NUM$ and NUM1$:

Example

```
10    A$ = NUM$(1000000)
      B$ = NUM1$(1000000)
30    PRINT LEN(A$); "/"; A$; "/"
      PRINT LEN(B$); "/"; B$; "/"
50    END
```

Output

```
8 / .1E+07 /
7 /1000000/
```

Note that *A$* has a leading and trailing space.

8.1.3.3 The VAL% and VAL Functions

The VAL% function returns the integer value of a numeric string. This numeric string expression must be the string representation of an integer. It can contain the ASCII digits 0 through 9 and the symbols + and –.

The VAL function returns the floating-point value of a numeric string. The numeric string expression must be the string representation of some number. It can contain:

- The ASCII digits 0 through 9
- The symbols +, – and .
- An uppercase E

VAL returns a number of the default floating-point data type. BASIC–PLUS–2 signals "Illegal number" (ERR = 52), if the argument is outside the range of the default floating-point data type.

The following is an example of VAL and VAL%:

Example

```
10    A = VAL("922")
20    B$ = "100"
30    C% = VAL%(B$)
40    PRINT A
50    PRINT C%
60    END
```

Output

```
922
100
```

8.1.4 Using String Arithmetic Functions

String arithmetic functions process numeric strings as arithmetic operands. This lets you add (SUM$), subtract (DIF$), multiply (PROD$) or divide (QUO$) numeric strings, or express them at a specified level of precision (PLACE$).

String arithmetic offers greater precision than floating-point arithmetic or longword integers, and it eliminates the need for scaling. However, string arithmetic executes much more slowly than the corresponding integer or floating-point operations.

The operands for the functions can be numeric strings representing any integer or floating-point value (E notation is not valid). Table 8–1 shows the string arithmetic functions and their formats, and gives brief descriptions of what they do. Later sections give more detailed descriptions of some of these string functions.

Table 8–1 String Arithmetic Functions

Function	Format	Description
SUM$	SUM$(A$,B$)	B$ is added to A$.
DIF$	DIF$(A$,B$)	B$ is subtracted from A$.
PROD$	PROD$(A$,B$,P%)	A$ is multiplied by B$. The product is expressed with precision P%.
QUO$	QUO$(A$,B$,P%)	A$ is divided by B$. The quotient is expressed with precision P%.
PLACE$	PLACE$(A$,P%)	A$ is expressed with precision P%.

SUM$ and DIF$ take the precision of the more precise argument in the function, unless padded zeros generate that precision. SUM$ and DIF$ omit trailing zeros to the right of the decimal point.

String arithmetic computations permit 56 significant digits. The functions QUO$, PLACE$, and PROD$, however, permit up to 60 significant digits. Table 8–2 shows how BASIC–PLUS–2 determines the precision permitted by each function and if that precision is implicit or explicit.

Table 8–2 Precision of String Arithmetic Functions

Function	How Determined	How Stated
SUM$	Precision of argument	Implicitly
DIF$	Precision of argument	Implicitly
PROD$	Value of argument	Explicitly
QUO$	Value of argument	Explicitly
PLACE$	Value of argument	Explicitly

The size and precision of results returned by the SUM$ and DIF$ functions depend on the size and precision of the arguments involved:

- The sum or difference of two integers takes the precision of the larger integer.
- The sum or difference of two decimal fractions takes the precision of the more precise fraction.
- The sum or difference of two real numbers takes precision as follows:
  - The sum or difference of the integer parts takes the precision of the larger part.

- The sum or difference of the decimal fraction parts takes the precision of the more precise part.

- BASIC–PLUS–2 truncates trailing zeros.

In the PLACE$, PROD$, and QUO$ functions, the value of the integer expression argument explicitly determines numeric precision. That is, the integer expression parameter determines the point at which the number is rounded or truncated.

If the integer expression is between –5000 and 5000, rounding occurs according to the following rules:

- For positive integer expressions, rounding occurs to the right of the decimal place. For example, if the integer expression is 1, rounding occurs one digit to the right of the decimal place (the number is rounded to the nearest tenth). If 2, rounding occurs two digits to the right of the decimal place (the number is rounded to the nearest hundredth), and so on.
- For zero, BASIC–PLUS–2 rounds to the nearest unit.
- For negative integer expressions, rounding occurs to the left of the decimal place. For example, if the integer expression is –1, rounding occurs one place to the left of the decimal point. In this case, BASIC–PLUS–2 moves the decimal point one place to the left, then rounds to units. If the integer expression is –2, rounding occurs two places to the left of the decimal point; BASIC–PLUS–2 moves the decimal point two places to the left, then rounds to units.

Note that when rounding numeric strings, BASIC–PLUS–2 returns only part of the number.

If the integer expression is between 5001 and 15000, the following rules apply:

- If the integer expression is 10000, BASIC–PLUS–2 truncates the number at the decimal point.
- If the integer expression is greater than 10000 (10000 plus *n*) BASIC–PLUS–2 truncates the numeric string *n* places to the right of the decimal point. For example, if the integer expression is 10001 (10000 plus 1), BASIC–PLUS–2 truncates the number starting one place to the right of the decimal point. If 10002 (10000 plus 2), BASIC–PLUS–2 truncates the number starting two places to the right of the decimal point, and so on.

- If the integer expression is less than 10000 (10000 minus *n*) BASIC–PLUS–2 truncates the numeric string *n* places to the left of the decimal point. For example, if the integer expression is 9999 (10000 minus 1), BASIC–PLUS–2 truncates the number starting one place to the left of the decimal point. If 9998 (10000 minus 2), BASIC–PLUS–2 truncates starting two places to the left of the decimal point, and so on.

For examples of this rounding and truncation behavior, see the following explanation of the PLACE$ function.

8.1.4.1 The PLACE$ Function

The PLACE$ function returns a numeric string, truncated or rounded according to an integer argument you supply.

The following example displays the use of the PLACE$ function with several different integer expression arguments.

Example

```
10    number$ = "123456.654321"
20    FOR I% = -5% TO 5%
         PRINT PLACE$(number$, I%)
      NEXT I%
30    PRINT
40    FOR I% = 9995 TO 10005
         PRINT PLACE$(number$, I%)
      NEXT I%
```

Output

```
1
12
123
1235
12346
123457
123456.7
123456.65
123456.654
123456.6543
123456.65432
```

```
1
12
123
1234
12345
123456
123456.6
123456.65
123456.654
123456.6543
123456.65432
```

8.1.4.2 The PROD$ Function

The PROD$ function returns the product of two numeric strings. The returned string's precision depends on the value you specify for the integer precision expression. (See Section 8.1.4 for allowable values of the integer precision expression).

Example

```
10    A$ = "-4.333"
20    B$ = "7.23326"
30    s_product$ = PROD$(A$, B$, 10005%)
40    PRINT s_product$
50    END
```

Output

```
-31.34171
```

8.1.5 Using Date and Time Functions

BASIC–PLUS–2 supplies functions to return the date and time in numeric or string format. The following sections discuss these functions.

8.1.5.1 The DATE$ Function

The DATE$ function returns a string containing a day, month, and year in the form *dd-Mmm-yy*. The date integer argument to the DATE$ function can have up to six digits in the form *yyyddd*, where *yyy* specifies the number of years since 1970 and *ddd* specifies the day of that year. If the numeric expression is zero, DATE$ returns the current date.

Example

```
10    PRINT DATE$(0)
20    PRINT DATE$(126)
30    PRINT DATE$(6168)
40    END
```

Output

```
15-Apr-91
06-May-70
16-Jun-76
```

If you supply an invalid date (for example, day 370 of 1973), the results are undefined.

8.1.5.2 The TIME$ Function

The TIME$ function returns a string displaying the time of day in the form *hh:mm* AM or *hh:mm* PM. TIME$ returns the time of day at a specified number of minutes before midnight. If you specify zero in the numeric expression, TIME$ returns the current time of day.

Example

```
10 PRINT TIME$(0)
20 PRINT TIME$(1)
30 PRINT TIME$(1440)
40 PRINT TIME$(721)
```

Output

```
01:53 PM
11:59 PM
12:00 AM
11:59 AM
```

8.1.5.3 The TIME Function

The TIME function requests time and usage information from the operating system and returns it to your program. The information returned by the TIME function depends on the value of the argument passed to it.

On RSTS/E systems, you can specify the following values as arguments to the TIME function:

0 Returns the number of seconds elapsed since midnight

1 Returns the current job's CPU time in tenths of a second

2 Returns the current job's connect time in seconds

3 Returns the number of kilo-core ticks (KCTs) that your job used

4 Returns the device time for the job in minutes

On RSX systems, you can only specify a value of zero with the TIME function.

On both RSX and RSTS/E systems, the following example prints the number of seconds that have elapsed since midnight:

Example

```
10  PRINT TIME(0)
20  END
```

Output

```
50755
```

8.1.6 Using Terminal Control Functions

BASIC–PLUS–2 provides several terminal control functions. These functions let you do the following:

- Enable and disable Ctrl/C trapping
- Enable and disable terminal echoing
- Read a single keystroke from a terminal

8.1.6.1 The CTRLC and RCTRLC Functions

The CTRLC function enables Ctrl/C trapping and the RCTRLC function disables Ctrl/C trapping. When Ctrl/C trapping is enabled, control is transferred to the program's error handler when a Ctrl/C is detected at the controlling terminal.

Ctrl/C trapping is asynchronous. The trap can occur in the middle of an executing statement, and a statement so interrupted leaves variables in an undefined state. For example, the statement A$ = "ABC", if interrupted by Ctrl/C, could leave the variable *A$* partially set to "ABC" and partially left with its old contents. Therefore, you should use the CTRLC function only when doing a final cleanup before exiting a program.

For example, if you enter a Ctrl/C to the following program when Ctrl/C trapping is enabled, an "ABORT" message prints to the file open on channel #1. This lets you know that the program did not end correctly.

Example

```
10   ON ERROR GOTO 50
     Y% = CTRLC
   .
   .
   .
50   IF ERR = 28
      THEN PRINT #1%, "Abort"
   .
   .
   .
```

8.1.6.2 The ECHO and NOECHO Functions

The NOECHO function disables echoing on a specified channel. Echoing is the process by which characters entered at the terminal keyboard appear on the terminal screen.

If you specify channel #0 (your terminal) as the argument, the characters entered on the keyboard are still accepted as input; however, they do not appear on the screen.

The ECHO function enables echoing on a specified channel and cancels the effect of the NOECHO function on that channel.

If you do not use these functions, ECHO is the default. This program shows a password routine in which the password does not echo:

Example

```
10    Y% = NOECHO(0%)
      INPUT "PASSWORD"; pword$
      IF pword$=="PLUGH" THEN PRINT "THAT IS CORRECT"
      END IF
20    Y% = ECHO(0%)
      END
```

Note that the Y% = ECHO(0%) statement is necessary to turn the echo back on. If this statement were not included, then all subsequent user inputs would not echo to the terminal screen.

8.2 User-Defined Functions

The DEF statement lets you create your own single-line or multi-line functions.

In the traditional BASIC–PLUS–2 usage, a function name consists of the following:

- The letters FN
- 1 to 28 letters, digits, underscores, or periods
- An optional percent sign or dollar sign

Integer function names must end with a percent sign and string function names must end with a dollar sign. Therefore, the function name can have up to 31 characters. If the function name ends with neither a percent sign nor a dollar sign, the function returns a real number.

You can still create user-defined functions using these function names. However, it is recommended that you use explicit data typing when defining functions for new program development. See Section 8.2.2 for an example of an explicitly declared function.

Note that the function name must start with FN only if the function is not explicitly declared and a function reference lexically precedes the function definition.

DEF functions can be either single-line or multi-line. Whether you use the single-line or multi-line format for function definitions depends on the complexity of the function you create. In general, multi-line DEF functions perform more complex functions than single-line DEF functions. However, the important distinction between single- and multi-line DEF functions is that multi-line DEF functions can be invoked recursively, whereas single-line DEF functions cannot.

If you want to pass values to a function, the function definition requires a formal parameter list. These formal parameters are the variables used to calculate the value returned by the function. When you invoke a function, you supply an actual parameter list; the values in the actual parameter list are copied into the formal parameter at this time. DEF functions allow up to 32 formal parameters. You can specify variables, constants, or array elements as formal parameters, but you cannot specify an entire array as a parameter to a DEF function.

8.2.1 Single-Line DEF Functions

In a single-line DEF, the function name, the formal parameter list, and the defining expression all appear on the same line. The defining expression specifies the calculations that the function performs. You can pass up to 32 arguments to this function through the formal parameter list. These parameters are variables local to the function definition, and each formal parameter can be preceded by a data type keyword.

The following example creates a function named *fnratio*. This function has two formal parameters: *numer* and *denomin*, whose ratio is returned as a REAL value.

When the function is invoked, BASIC–PLUS–2 does the following:

- Copies the values 5.6 and 7.8 into the formal parameters *numer* and *denomin*
- Evaluates the expression to the right of the equal sign
- Returns the value to the statement that invoked the function (the PRINT statement)

The PRINT statement then prints the returned value.

Example

```
10     DEF REAL fnratio (numer, denomin) = numer / denomin
20     PRINT fnratio(5.6, 7.8)
30     END
```

Output

```
.717949
```

Note that the actual parameters you supply must agree in number and data type with those in the formal parameter list; you must supply numeric values for numeric variables, and string values for string variables.

The defining expression for a single-line function definition can contain any constant, variable, BASIC–PLUS–2 built-in function, or any user-defined function except the function being defined. The following are valid function definitions:

Example

```
10     DEF FN_A(X) = X^2 + 3 * X + 4
20     DEF FN_B(X) = FN_A(X) / 2 + FN_A(X)
30     DEF FN_C(X) = SQR(X+4) + 1
40     DEF CUBE(X) = X ^ 3
```

Note that the name of the last function defined does not begin with FN. This is valid as long as no reference to the function lexically precedes the function definition.

You can also define a function that has no formal parameters. For instance, the following function definition uses three BASIC–PLUS–2 built-in functions to return an integer corresponding to the day of the month. DATE$(0) returns a date string in the form *dd-Mmm-yy*. The SEG$ function strips out of this value the characters starting at character position one up to and including the character at position two (the day number). The VAL% function converts this resulting numeric string to an integer. In this way, *fnday_number* returns the day of the month as an integer.

```
10     DEF INTEGER fnday_number = VAL% (SEG$(DATE$(0%), 1%, 2%))
```

8.2.2 Multi-Line DEF Functions

The DEF statement can also define multi-line functions. Multi-line DEF functions are useful for expressing complicated functions. Note that multi-line DEF functions do not have the equal sign and defining expression on the first line. Instead, this expression appears in the function block, assigned to the function name.

Note

If a multi-line DEF function contains DATA statements, they are global to the program.

Multi-line function definitions can contain any constant, variable, BASIC–PLUS–2 built-in function, or user-defined function.

You can use either the END DEF or EXIT DEF statements to exit from a user-defined function. The EXIT DEF statement is equivalent to an unconditional transfer to the END DEF statement.

The following example shows a multi-line DEF function that uses both the EXIT and END DEF statements. The defining expression of the function is in the ELSE clause. This assigns a value to the function if *A* is less than 10. The second set of output shows what happens when *A* is greater than 10; BASIC–PLUS–2 prints the message "Out of range" and executes the EXIT DEF statement. The function returns zero because control is transferred to the END DEF statement before a value was assigned. In this way, this example tests the arguments before the function is evaluated.

Example

```
10     DEF fn_discount(A)
20        IF A > 10
30          THEN
              PRINT "OUT OF RANGE"
              EXIT DEF
40          ELSE
              fn_discount = A^A
50        END IF
60     END DEF

70     INPUT Z
80     PRINT fn_discount(Z)
90     END
```

Output 1

```
? 4
256
```

Output 2

```
? 12
OUT OF RANGE
0
```

If you do not explicitly declare the function with the DECLARE statement, the restrictions for naming a multi-line DEF function are the same as those for the single-line DEF function. However, explicitly declaring a DEF function can make a program easier to read and understand. For instance, in the following examples, the first example concatenates two strings and the second returns a number in a specified module:

Example 1

```
10     DECLARE STRING FUNCTION concat (STRING, STRING) !Declare the function
   .
   .
   .
100    DEF STRING concat (STRING Y, STRING Z)
120    concat = Y + Z  !Define the function
140    FNEND
   .
   .
   .
180    new_string$ = concat(A$, B$) !Invoke the function
   .
   .
   .
210    END
```

Example 2

```
10     DECLARE REAL FUNCTION mdlo (REAL, INTEGER)
20     DEF mdlo( REAL  argument, INTEGER modulus )
       !Check for argument equal to zero

30     EXIT DEF IF argument = 0
       !Check for modulus equal to zero, modulus equal to absolute
       !value of argument, and modulus greater than absolute
       !value of argument.

40     SELECT modulus
         CASE = 0%
           EXIT DEF
         CASE > ABS( argument )
           EXIT DEF
         CASE = ABS( argument )
           mdlo = argument
           EXIT DEF
50     END SELECT

       !If argument is negative, set flag negative% and set argument
       !to its absolute value.
60     IF argument < 0
          THEN argument = ABS( argument )
             negative%  = -1%
70     END IF
80     UNTIL argument < modulus
         argument = argument - modulus
```

```
        !If this calculation ever results in zero, mdlo returns zero
90          IF argument = modulus
                THEN mdlo = 0
                EXIT DEF
100         END IF
110     NEXT

        !Argument now contains the right number, but the sign may be wrong.
        !If the negative argument flag was set, make the result negative.

120     IF negative%
            THEN mdlo = - argument
            ELSE mdlo = argument
130     END IF

140     END DEF

200     INPUT "PLEASE INPUT THE VALUE AND THE MODULUS"; X,Y
210     PRINT mdlo(X,Y)
220     END
```

Output

```
PLEASE INPUT THE VALUE AND THE MODULUS? 7, 5
2
```

Because these functions are declared in DECLARE statements, the function names do not have to conform to the traditional BASIC–PLUS–2 rules for naming functions.

Recursion occurs when a function calls itself. The following example defines a recursive function that returns a number's factorial value:

Example

```
10      DECLARE INTEGER FUNCTION factor ( INTEGER )
        DEF INTEGER factor ( INTEGER F )
           IF F <= 0%
              THEN factor = 1%
              ELSE factor = factor(F - 1%) * F
           END IF
        END DEF
20      INPUT "INPUT N TO FIND N FACTORIAL"; N%
        PRINT "N! IS"; factor(N%)
30      END
```

Output

```
INPUT N TO FIND N FACTORIAL? 5
N! IS 120
```

Any variable accessed or declared in the DEF function and not in the formal parameter list is global to the program unit. When BASIC–PLUS–2 evaluates the user-defined function, these global variables contain the values last assigned to them in the surrounding program module.

To prevent confusion, variables declared in the formal parameter list should not appear elsewhere in the program. Note that if your function definition actually uses global variables, these variables cannot appear in the formal parameter list.

You cannot transfer control into a multi-line DEF function except by invoking it. You should not transfer control out of a DEF function except by way of an EXIT DEF or END DEF statement. This means the following:

- If the DEF function contains an ON ERROR GOTO, GOTO, ON GOTO, GOSUB, ON GOSUB, or RESUME statement, that statement's target line number must also be in that DEF function.
- An ON ERROR GO BACK statement can transfer control out of a DEF function; however, a RESUME statement in an error handler outside the DEF function cannot transfer control back into the DEF function.
- A subroutine cannot be shared by more than one DEF function. However, if you rewrite the subroutine as a DEF function with no parameters, other function definitions can share it.

A DEF function never changes the value of a parameter passed to it. Also, because formal parameters are local to the function definition, you cannot access the values of these variables from outside the DEF statement. These variable names are known only inside the DEF statement.

In the following example, the variable *first* is declared only in the function *fn_sum*. When BASIC–PLUS–2 sees the PRINT *first* statement, it assumes that *first* is a new variable that is not declared in the main program. If you try to run this example, BASIC–PLUS–2 signals the error "explicit declaration of first required." If you do not specify the OPTION TYPE = EXPLICIT statement, BASIC–PLUS–2 assumes that *first* is a new variable and initializes it to zero.

Example

```
10    OPTION TYPE = EXPLICIT
      DECLARE INTEGER A, B
      DEF fn_sum(INTEGER first, INTEGER second) = first + second

      A = 50%
      B = 25%

30    PRINT fn_sum(A, B)
40    PRINT first
50    END
```

8.3 External Functions

An external function is a separately compiled program module that returns a value. To create the function subprogram, you use the FUNCTION, END FUNCTION, and EXIT FUNCTION statements. The difference between a FUNCTION subprogram and a SUB subprogram is that the FUNCTION subprogram returns a value.

External functions are useful because they do the following:

- Can be invoked by any program module
- Allow up to 32 parameters
- Allow arrays to be passed as parameters
- Allow more than one value to be returned by modifiable parameters

You use the EXTERNAL statement to name and explicitly declare the data type returned by the external function and the data type of the formal parameters.

After you have created an external function, you compile it and task-build it with the program modules that invoke it. The syntax for invoking an external function is the same as for invoking a built-in or DEF function.

8.3.1 The FUNCTION, EXIT FUNCTION, and END FUNCTION Statements

The FUNCTION statement marks the beginning of a FUNCTION subprogram. The END FUNCTION statement does the following:

- Marks the end of a function subprogram
- Returns a value
- Returns program control to the statement that invoked the function

FUNCTIONEND is a synonym for END FUNCTION, but END FUNCTION is preferred.

The EXIT FUNCTION statement immediately returns program control to the statement that invoked the function. It is equivalent to an unconditional transfer to the FUNCTIONEND statement. FUNCTIONEXIT is a synonym for EXIT FUNCTION but, EXIT FUNCTION is preferred. The following is an example of an external function:

Example

```
100     FUNCTION REAL volume (REAL R)
200     IF R <= 0 THEN EXIT FUNCTION
300     volume = 4/3 * PI * R ** 3
400     END FUNCTION
```

This function returns the volume of a sphere of radius R. If this function is invoked with an actual parameter value less than or equal to zero, the function returns zero.

Because external functions are subprograms, you can pass modifiable parameters (including entire arrays) to them.

See Chapter 11 for more information on subprograms and modifiable parameters.

8.3.2 The EXTERNAL Statement

Function subprograms must be declared with the EXTERNAL statement. The following example shows the declaration and invocation of the external function as defined in the example in the previous section:

Example

```
1000    EXTERNAL REAL FUNCTION volume (REAL)
1100    temp_volume = volume(5.925)
1200    PRINT temp_volume
```

Note that the EXTERNAL statement specifies only the data type of the parameter; it does not specify a formal or actual parameter. Whenever you invoke an external function, BASIC–PLUS–2 converts the actual parameter to the data type specified in the EXTERNAL statement.

To run this program in the BASIC environment, follow these steps:

1. Compile the function subprogram
2. Load the resulting object module with the LOAD command
3. Read in the main program with the OLD command
4. Enter the RUN command

Note that rather than running this program interactively, you can task-build these modules together and run the program from DCL level. See Chapter 11 for more information on task-building subprograms.

Note

Ensure that the parameter data types specified in the EXTERNAL statement agree with those specified in the FUNCTION statement; otherwise, the function will produce unexpected results.

9 String Handling

This chapter defines dynamic, and fixed-length strings as well as string virtual arrays, explains which you should choose for your application, and shows you how to use them.

9.1 Introduction

A string is a sequence of ASCII characters. BASIC–PLUS–2 allows you to use three types of strings:

- Dynamic strings
- Fixed-length strings
- String virtual arrays

Dynamic strings are strings whose length can change during program execution. The length of a dynamic string variable may or may not change, depending on the statement used to modify it.

Fixed-length strings are strings whose length never changes. In other words, their length remains static. String constants are always fixed-length. String variables can be either fixed-length or dynamic. When a string variable is fixed-length, its length does not change, regardless of the statement you use to modify it. See Table 9–1 for more information on string modification.

Strings in virtual arrays have both fixed-length and dynamic attributes. That is, string virtual arrays have a specified maximum length between 0 and 512 characters. During program execution, the length of an element in a string virtual array can change; however, the length is always between zero and the maximum string size specified when the array was created. See Section 9.4 and Chapter 12 for more information about virtual arrays.

Table 9–1 String Modification

Statement	For Fixed-Length Strings, Changes	For Dynamic Strings, Changes	For Virtual Array Strings, Changes
LET	Value	Value and Length	Value and Length
LSET	Value	Value	Value and Length
RSET	Value	Value	Value and Length
Terminal I/O Statements[1]	Value	Value and Length	Value and Length

[1]Terminal I/O statements include INPUT, INPUT LINE, LINPUT, MAT INPUT, and so on.

9.2 Using Dynamic Strings

Although dynamic strings are less efficient than fixed-length strings, they are often more flexible. For example, to concatenate strings, you can just use the LET statement to assign the concatenated value to a dynamic string variable, without having to worry about BASIC–PLUS–2 truncating the string or adding trailing spaces to it. However, if the destination variable is fixed-length, you must make sure that it is long enough to receive the concatenated string, or BASIC–PLUS–2 truncates the new value to fit the destination string. Similarly, if you use LSET or RSET to concatenate strings, you must ensure that the destination variable is long enough to receive the data.

The LET, LSET and RSET statements all operate on dynamic strings as well as fixed-length strings. The LET statement can change the length of a dynamic string; however, LSET and RSET do not.

In the following example, the first line assigns the value "ABC" to *A$*, the second line assigns "XYZ" to *B$*, and the third line assigns six spaces to *C$*. Each of these variables are dynamic strings. In the fourth line, LSET assigns *A$* the value of *A$* concatenated with *B$*. Because the LSET statement does not change the length of the destination string variable, only the first three characters of the expression *A$* + *B$* are assigned to *A$*. The fifth line uses LSET to assign *C$* the value of *A$* concatenated with *B$*. Because *C$* already has a length of six, this statement assigns the value ABCXYZ to it.

Example

```
10     LET A$ = "ABC"
20     LET B$ = "XYZ"
30     LET C$ = "      "
40     LSET A$ = A$ + B$
50     LSET C$ = A$ + B$
60     PRINT A$
70     PRINT C$
80     END
```

Output

```
ABC
ABCXYZ
```

Like the LET statement, the INPUT, INPUT LINE, and LINPUT statements can change the length of a dynamic string, but they cannot change the length of a fixed-length string.

In the next example, the first line assigns the null string to variable *A$*. The second line uses the LEN function to show that the null string has a length of zero. The third line uses the INPUT statement to assign a new value to *A$*, and the fourth and fifth lines print the new value and its length.

Example

```
10     !Declare a dynamic string
20     LET A$ = ""
30     PRINT LEN(A$)
40     INPUT A$
50     PRINT A$
60     PRINT LEN(A$)
70     END
```

Output

```
0
? THIS IS A TEST
THIS IS A TEST
14
```

You should not confuse the null string with a null character. A null character is one whose ASCII numeric code is zero. The null string is a string whose length is zero.

9.3 Using Fixed-Length Strings

It is more efficient to change a fixed-length string than a dynamic string. Creating or modifying a dynamic string often causes BASIC–PLUS–2 to create new storage, and this increases processor overhead. Modifying fixed-length strings involves less overhead because BASIC–PLUS–2 simply reuses existing storage.

If a string variable is part of a MAP, COMMON, or virtual array, a LET, INPUT, LINPUT, or INPUT LINE statement changes its value, but not its length.

In the following example, the MAP statement in the first line explicitly assigns a length to each string variable. Because the LINPUT statements cannot change this length, BASIC–PLUS–2 truncates values to fit the *address* and *city_state* variables. Because the *zip* variable is longer than the assigned value, BASIC–PLUS–2 left-justifies the assigned value and pads it with spaces. The sixth line uses the compile-time constant HT (horizontal tab) to separate fields in the employee record.

Example

```
10     MAP (FIELDS) STRING full_name = 10,                          &
                   address = 10,                                    &
                   city_state = 10,                                 &
                   zip = 10
20     LINPUT "NAME"; full_name
       LINPUT "ADDRESS"; address
       LINPUT "CITY AND STATE"; city_state
       LINPUT "ZIP CODE"; zip
40     EMPLOYEE_RECORD$ = full_name + HT + address + HT &
                          + city_state + HT + zip
50     PRINT EMPLOYEE_RECORD$
60     END
```

Output

```
NAME? JOE SMITH
ADDRESS? 66 GRANT AVENUE
CITY AND STATE? NEW YORK NY
ZIP? 01001

JOE SMITH          66 GRANT A   NEW YORK N 01001
```

9.4 String Virtual Arrays

Virtual arrays are stored on disk. You create a virtual array by opening a disk file and then using the DIM # statement to dimension the array on the open channel. This section describes only string virtual arrays. See Chapter 12 for more information on virtual arrays.

Elements of string virtual arrays behave much like dynamic strings, except that for the following:

- When you create the virtual string array, you specify a maximum length for the array's elements. The length of an array element can never exceed this maximum. If you do not supply a length, the default is 16 characters.
- A string virtual array element cannot contain trailing nulls.

When you assign a value to a string virtual array element, BASIC–PLUS–2 pads the value with nulls, if necessary, to fit the length of the virtual array element. However, when you retrieve the virtual array element, BASIC–PLUS–2 strips all trailing nulls from the string. Therefore, when you access an element in a string virtual array, the string never has trailing nulls.

In the following example, the first two lines dimension a string virtual array and open a file on channel #1. The third line assigns a 10-character string to the first element of this string array, and to the variable *A$*. This 10-character string consists of "ABCDE" plus five null characters. The PRINT statements show that the length of *A$* is 10, while the length of *test(1)* is only 5 because BASIC–PLUS–2 strips trailing nulls from string array elements.

Example

```
10     DIM #1%, STRING test(5)
20     OPEN "TEST" AS FILE #1%, ORGANIZATION VIRTUAL
30     A$, test(1%) = "ABCDE" + STRING$(5%, 0%)
40     PRINT "LENGTH OF A$ IS: "; LEN(A$)
50     PRINT "LENGTH OF TEST(1) IS: "; LEN(test(1%))
60     END
```

Output

```
LENGTH OF A$ IS: 10
LENGTH OF TEST(1) IS: 5
```

Although the storage for string virtual array elements is fixed, the length of a string array element can change because BASIC–PLUS–2 strips the trailing nulls whenever it retrieves a value from the array.

9.5 Assigning String Data

To assign string data, you use the LET, LSET, and RSET statements. The following sections describe how to incorporate each of these statements into your source code.

9.5.1 The LET Statement

The LET statement assigns string data to a string variable. The keyword LET is optional. In the following example, *B* is a string variable and "ret_status" is a quoted string expression.

```
10    LET B = "ret_status"
```

The LET statement changes the length of dynamic strings but does not change the length of fixed-length strings. For instance, the following example first creates a fixed-length string named *ABC* by declaring the string in a MAP statement. The program then creates a dynamic string named *XYZ* by declaring it in a DECLARE statement. The third line assigns a 3-character value to both variables *ABC* and *XYZ*, then prints the value and the length of the string variables. Variable *ABC* continues to have a length of 10: the three characters assigned, plus seven spaces for padding. The length of the dynamic variable changes with the values assigned to it.

Example

```
10    MAP (TEST) STRING ABC = 10
20    DECLARE STRING XYZ
30    ABC = "ABC"
40    XYZ = "XYZ"
50    PRINT ABC, LEN(ABC)
60    PRINT XYZ, LEN(XYZ)
70    ABC = "A"
80    XYZ = "X"
90    PRINT ABC, LEN(ABC)
100   PRINT XYZ, LEN(XYZ)
```

Output

```
ABC           10
XYZ           3
A             10
X             1
```

9.5.2 The LSET Statement

The LSET statement left-justifies data and assigns it to a string variable, without changing the variable's length. In the following example, *ABC* is a string variable and "ABC" is a string constant.

```
10    LSET ABC = "ABC"
```

If the string expression's value is shorter than the string variable's current length, LSET left-justifies the expression and pads the string variable with spaces. In the following example, the LET statement creates the 5-character string variable *test$*. The LSET statement in the second line assigns the string XYZ to the variable *test$* but does not change the length of *test$*. Because *test$* has a length of five, the LSET statement pads the string XYZ with two spaces when assigning the value. The PRINT statement shows that test$ includes these two spaces.

Example

```
10    LET test$ = "ABCDE"
20    LSET test$ = "XYZ"
30    PRINT "'"; test$; "'"
40    END
```

Output

```
'XYZ  '
```

LSET left-justifies a string expression longer than the string variable and truncates it on the right as shown in the following example:

Example

```
10    LET test$ = "ABCDE"
      LSET test$ = "12345678"
      PRINT test$
      END
```

Output

```
12345
```

The LET statement creates the 5-character string variable *test$*. The LSET statement in the second line assigns the characters "12345" to *test$*. Because LSET does not change the string variable's length, it truncates the last three characters (678).

9.5.3 The RSET Statement

The RSET statement right-justifies data and assigns it to a string variable without changing the variable's length. In the following example, *C_R* is a string variable and "cust_rec" is a string constant.

```
10    RSET C_R = "cust_rec"
```

RSET right-justifies a string expression shorter than the string variable and pads it with spaces on the left. In the following example, the LET statement creates the 5-character string variable *test$*. The RSET statement in the second line assigns the string XYZ to *test$* but does not change the length of *test$*. Because *test$* is five characters long, the RSET statement pads XYZ with two spaces when assigning the value. The PRINT statement shows that *test$* includes these two spaces.

Example

```
10    LET test$ = "ABCDE"
      RSET test$ = "XYZ"
      PRINT "'" +  test$; "'"
20    END
```

Output

```
'  XYZ'
```

If the string expression's value is longer than the string variable, RSET right-justifies the string expression and truncates characters on the left to fit the string variable as shown in the following example:

Example

```
10    LET test$ = "ABCDE"
      RSET test$ = "987654321"
      PRINT test$
20    END
```

Output

```
54321
```

The LET statement creates a 5-character string variable, *test$*. The RSET statement assigns "54321" to *test$*. RSET, which does not change the variable's length, truncates "9876" from the left side of the string expression.

Note that, when using LSET and RSET, padding can become part of the data:

Example

```
10    LET A$ = '12345'
      LSET A$ = 'ABC'
      LET B$ = '12345678'
      RSET B$ = A$
      PRINT "'";B$;"'"
```

Output

```
'    ABC   '
```

9.6 Manipulating String Data with String Functions

When used with the LET statement, BASIC–PLUS–2 string functions let you manipulate and modify strings. These functions let you do the following:

- Determine the length of a string (LEN)
- Search for the position of a set of characters in a string (POS)
- Extract segments from a string (SEG$, MID$)
- Substitute string data in a portion of a string variable (MID$)
- Create a string of any length, made up of any single character (STRING$)
- Create a string of spaces (SPACE$)
- Remove trailing spaces and tabs from a string (TRM$)
- Edit a string (EDIT$)

9.6.1 The LEN Function

The LEN function returns the number of characters in a string as an integer value. For example:

```
10     LEN(spec)
```

Spec is a string expression. The length of the string expression includes leading and trailing blanks. In the following example, the variable Z$ is set equal to "ABC XYZ", which has a length of eight.

Example

```
10     alpha$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
       PRINT LEN(alpha$)
       Z$ = "ABC" + "  " + ""XYZ"
       PRINT LEN(Z$)
20     END
```

Output

```
26
8
```

9.6.2 The POS Function

POS searches a string for a group of characters (a substring). In the following example, *spec* is the string to be searched, *test* is the substring for which you are searching and *15* is the character position where BASIC–PLUS–2 starts the search.

```
10      POS(spec,test,15)
```

The position returned by POS is relative to the beginning of the string, not the starting position of the search. For example, if you search the string "ABCDE" for the substring "E", it does not matter whether you specify a starting position of one, two, three, four, or five BASIC–PLUS–2 still returns the value five as the position where the substring was found.

If the function finds the substring, it returns the position of the substring's first character. Otherwise, it returns zero as in the following example:

Example

```
10      alpha$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        Z$ = "DEFG"
        X% = POS(ALPHA$,Z$,1%)
20      PRINT X%
        Q$ = "TEST"
        Y% = POS(ALPHA$, Q$, 1%)
30      PRINT Y%
40      END
```

Output

```
4
0
```

If you specify a starting position other than one, BASIC–PLUS–2 still returns the position of the substring relative to the beginning of the string as shown in the following example:

Example

```
10      alpha$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        Z$ = "HIJ"
20      PRINT POS(ALPHA$, Z$, 7%)
30      END
```

Output

```
8
```

If you know that the substring is not near the string's beginning, specifying a starting position greater than one speeds program execution by reducing the number of characters BASIC–PLUS–2 must search.

You can use the POS function to associate a character string with an integer that you can then use in calculations. This technique is called a *table look-up*. For instance, the following example prompts for a 3-character string, changes the string to uppercase letters and searches the table string to find a match. The WHILE loop executes indefinitely until a carriage return is entered in response to the prompt.

Example

```
10    DECLARE STRING CONSTANT table =    &
       "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC"
      DECLARE STRING month, UPPER_CASE_MONTH, message
      DECLARE INTEGER month_length
      DECLARE REAL month_pos
20    PRINT "Please type the first three letters of a month"
      PRINT "To end the program, type only RET";
30    Loop_1:
40    WHILE 1% = 1%
         INPUT month
         UPPER_CASE_MONTH = EDIT$(month, 32%)
         month_length = LEN(UPPER_CASE_MONTH)
         EXIT Loop_1 IF month_length = 0%
         IF month_length = 3%
            THEN month_pos = (POS(table, UPPER_CASE_MONTH, 1) + 2) / 3
                IF (month_pos = 0%) OR (month_pos <> FIX(month_pos))
                   THEN MESSAGE = "Invalid abbreviation, try again"
                   ELSE MESSAGE = "is month number" + NUM$(MONTH_POS)
                END IF
            ELSE MESSAGE = "Abbreviation not three characters, try again"
50       END IF
60       PRINT month; message
70    NEXT
80    END
```

Output

```
Please type the first three letters of a month
To end the program, type only RET? Nov
Nov is month number 11
```

Keep these considerations in mind when you use the POS function:

- If you specify a starting position less than one, POS assumes a starting position of one.
- If you specify a starting position greater than the searched string's length, POS returns a zero (unless the substring is null).
- When searching for a null string:
  - If you specify a starting position greater than the string's length, POS returns the string's length plus one.

- If the string to be searched is also null, POS returns a value of one.
- If the specified starting position is less than or equal to one, POS returns a value of one.
- If the specified starting position is greater than one and less than or equal to the string's length plus one, POS returns the specified starting position.

Note that searching for a null string is not the same as searching for the null character. A null string has a length of zero, while the null character has a length of one. The null character is an ASCII character whose value is zero.

9.6.3 The SEG$ Function

The SEG$ function extracts a segment (substring) from a string. The original string remains unchanged. In the following example, *time* is the input string, *13* is the position of the first character extracted and *16* is the position of the last character extracted.

```
10    SEG$(time,13%,16%)
```

SEG$ extracts from the input string the substring that starts at the first character position, up to and including the last character position. It returns the extracted segment.

Example

```
10    PRINT SEG$("ABCDEFG", 3%, 5%)
20    END
```

Output

```
CDE
```

If you specify character positions that exist in the string, the length of the returned substring always equals (int-exp2–int-exp1 + 1).

Keep these considerations in mind when you use the SEG$ function:

- If the starting character position is less than one, BASIC–PLUS–2 assumes a value of one.
- If the starting character position is greater than the ending character position, or the length of the string, SEG$ returns a null string.
- If the ending character position is greater than the length of the string, SEG$ returns all characters from the starting character position to the end of the string.
- If the starting character position is equal to the ending character position, SEG$ returns the character at the starting position.

You can replace part of a string by using the SEG$ function with the string concatenation operator (+). In the following example, when BASIC–PLUS–2 creates *C$*, it concatenates the first two characters of *A$*, the 3-letter string XYZ, and the last two characters of *A$*. The original contents of *A$* do not change.

Example

```
10    A$ = "ABCDEFG"
20    C$ = SEG$(A$, 1%, 2%) + "XYZ" + SEG$(A$, 6%, 7%)
30    PRINT C$
40    PRINT A$
50    END
```

Output

```
ABXYZFG

ABCDEFG
```

You can use similar string expressions to replace characters in any string. A general formula to replace characters in positions *n* through *m* of string A$ with characters in B$ is as follows:

C$ = SEG$(A$,1%,n–1) + B$ + SEG$(A$,m+1,LEN(A$))

The following example replaces the sixth through ninth characters of the string "ABCDEFGHIJK" with "123456":

Example

```
10    A$ = "ABCDEFGHIJK"
      B$ = "123456"
      C$ = SEG$(A$,1%,5%) + B$ + SEG$(A$,10%,LEN(A$))
20    PRINT C$
30    PRINT A$
40    PRINT B$
50    END
```

Output

```
ABCDE123456JK
ABCDEFGHIJK
123456
```

The following formulas are more specific applications of the previous formula:

- To replace the first n characters of A$ with B$ use the following:

  C$ = B$ + SEG$(A$,n+1,LEN(A$))

- To replace all but the first n characters of A$ with B$ use the following:

  C$ = SEG$(A$,1,n) + B$

- To replace all but the last n characters of A$ with B$ use the following:

C$ = B$ + SEG$(A$,(LEN(A$)–n) + 1, LEN(A$))

- To replace the last n characters of A$ with B$ use the following:

  C$ = SEG$(A$,1,LEN(A$)–n) + B$

- To insert B$ in A$ after the nth character in A$ use the following:

  C$ = SEG$(A$,1,n) + B$ + SEG$(A$,n+1,LEN(A$))

9.6.4 The MID$ Function

The MID$ function *extracts* a specified substring, beginning at a specified character position and ending at a specified length. If you specify a starting character position that is less than one, BASIC–PLUS–2 automatically assumes a starting character position of one.

In the following example, the MID$ function uses the input string "ABCD," and extracts a segment consisting of 3 characters. Because BASIC–PLUS–2 automatically assumes a starting character position of one when the specified starting character position is less than one, the string that is extracted begins with the first character of the input string.

Example

```
10  DECLARE STRING old_string, new_string
20  old_string = "ABCD"
30  new_string = MID$(old_string,0%,3%)
40  PRINT new_string
```

Output

```
ABC
```

Keep these considerations in mind when you use the MID$ function:

- If the position of the segment's first character is greater than the input string, MID$ returns a null string.
- If the length of the segment is greater than the length of the input string, BASIC–PLUS–2 returns the string that begins at the specified starting character position and includes all characters remaining in the string.
- If the length of the segment is less than or equal to zero, MID$ returns a null string.
- If you specify a floating-point expression for the position of the segment's first character or for the length of the segment, BASIC–PLUS–2 truncates it to a WORD integer.

9.6.5 The STRING$ Function

The STRING$ function creates a character string containing multiple occurrences of a single character. In the following example, *23* is the length of the returned string and *30* is the ASCII value of the character that makes up the string. This value is treated modulo 256.

```
10    STRING$(23,30)
```

The following example creates a 10-character string containing uppercase As, which have ASCII value 65.

Example

```
10    out$ = STRING$(10%, 65%)
20    PRINT out$
30    END
```

Output

```
AAAAAAAAAA
```

Keep these considerations in mind when you use the STRING$ function:

- If the length of the returned string is less than or equal to zero, STRING$ returns a null string.
- If the length of the returned string is greater than 32767, BASIC–PLUS–2 signals an error.

9.6.6 The SPACE$ Function

The SPACE$ function creates a character string containing spaces. In this example, *5* is the number of spaces in the string.

```
10    SPACE$(5%)
```

The following example creates a 9-character string which contains 3 spaces.

Example

```
10    A$ = "ABC"
      B$ = "XYZ"
20    PRINT A$ + SPACE$(3%) + B$
30    END
```

Output

```
ABC   XYZ
```

9.6.7 The TRM$ Function

The TRM$ function removes trailing blanks and tabs from a string. The input string remains unchanged. In the following example, all trailing blanks that appear in the string expression, "ABCDE " are removed once the TRM$ function is invoked.

Example

```
10    A$ = "ABCDE    "
      B$ = "XYZ"
      first$ = A$ + B$
      second$ = TRM$(A$) + B$
20    PRINT first$
30    PRINT second$
40    END
```

Output

```
ABCDE    XYZ
ABCDEXYZ
```

The TRM$ function is especially useful for extracting the nonblank characters from a fixed-length string (for example, a COMMON or MAP, or a parameter passed from a program written in another language).

9.6.8 The EDIT$ Function

The EDIT$ function performs one or more string editing functions, depending on the value of an argument you supply. The input string remains unchanged. In the following example, *stu_rec* is a string expression and *32* determines the editing function performed.

```
10   EDIT$(stu_rec,32%)
```

Table 9–2 shows the action BASIC–PLUS–2 takes for a given value of int-exp.

Table 9–2 EDIT$ Options

Value of Int-exp	Effect
1	Discards each character's parity bit (bit 7). Note that you should not use this value for characters in the DEC Multinational Character Set.
2	Discards all spaces and tabs.
4	Discards all carriage returns, line feeds, form feeds, deletes, escapes, and nulls.

(continued on next page)

Table 9–2 (Cont.) EDIT$ Options

Value of Int-exp	Effect
8	Discards leading spaces and tabs.
16	Converts multiple spaces and tabs to a single space.
32	Converts lowercase letters to uppercase.
64	Converts left brackets ([ ) to left parentheses (( ), and right brackets ( ]) to right parentheses ( )).
128	Discards trailing spaces and tabs (same as TRM$ function).
256	Suppresses all editing for characters within quotation marks. If the string has only one quotation mark, BASIC–PLUS–2 suppresses all editing for the characters following the quotation mark (256% can also be specified as –1%).

All values are additive; for example, by specifying 168, you can do the following:

- Discard leading spaces and tabs (value 8)
- Convert lowercase letters to uppercase (value 32)
- Discard trailing spaces and tabs (value 128)

However, when specifying more than one EDIT$ value, you do not need to add the values together ahead of time. Instead, you can separate more than one EDIT$ value with a plus sign (+). This is easier and prevents human error. For example:

Example

```
10    LINPUT "PLEASE TYPE A STRING";input_string$
      new_string$ = EDIT$(input_string$, 2% + 32% + 64%)
20    PRINT new_string$
30    END
```

Output

```
PLEASE TYPE A STRING? 88  abc TAB [5,5]
88ABC(5,5)
```

This program requests an input string, discards all spaces and tabs, converts lowercase letters to uppercase, and converts brackets to parentheses.

9.7 Manipulating String Data with Multiple Maps

Mapping a string storage area in more than one way lets you extract a substring from a string or concatenate strings. In the following example, the three MAP statements reference the same 108 bytes of data.

Example

```
10    MAP (emprec) first_name$ = 10,              &
                   last_name$ = 20,               &
                   street_number$ = 6,            &
                   street$ = 15,                  &
                   city$ = 20,                    &
                   state$ = 2,                    &
                   zip$ = 5,                      &
                   wage_class$ = 2,               &
                   date_of_review$ = 8,           &
                   salary_ytd$ = 10,              &
                   tax_ytd$ = 10
20    MAP (emprec) full_name$ = 30,               &
                   address$ = 48,                 &
                   salary_info$ = 30
30    MAP (emprec) employee_record$ = 108
```

You can move data into a MAP in different ways. For instance, you can use terminal input, arrays, and files. In the following example, the READ and DATA statements are used to move data into a MAP:

Example

```
10    READ EMPLOYEE_RECORD$
100   DATA "WILLIAM   DAVIDSON            2241  MADISON BLVD    " &
      "SCRANTON            PA14225A912/10/78$13,325.77$925.31"
```

Because all the MAP statements in the previous example reference the same storage area *emprec*, you can access parts of this area through the mapped variables shown in the following two examples.

Example 1

```
10    PRINT full_name$
      PRINT wage_class$
      PRINT salary_ytd$
```

Output 1

```
WILLIAM   DAVIDSON
A9
$13,325.77
```

Example 2

```
10    PRINT last_name$
      PRINT tax_ytd$
```

Output 2

```
DAVIDSON
$925.31
```

You can assign a new value to any of the mapped variables. For instance, the following example prompts the user for changed information by displaying a menu of topics. The user can then choose which topics need to be changed and then separately assign new values to each variable.

Example

```
10    Loop_1:
        WHILE 1% = 1%
           INPUT "Changes? (please type YES or NO)"; CH$
           EXIT Loop_1 IF CH$ = "NO"
           PRINT "1. FIRST NAME"
           PRINT "2. LAST NAME"
           PRINT "3. STREET NUMBER"
           PRINT "4. STREET"
           PRINT "5. CITY"
           PRINT "6. STATE"
           PRINT "7. ZIP"
           PRINT "8. WAGE CLASS"
           PRINT "9. DATE OF REVIEW"
           PRINT "10. SALARY YTD"
           PRINT "11. TAX YTD"
           INPUT "CHANGE NUMBER"; NUMBER%
           SELECT NUMBER%
              CASE 1%
                 INPUT "FIRST NAME"; first_name$
              CASE 2%
                 INPUT "LAST NAME"; last_name$
              CASE 3%
                 INPUT "STREET NUMBER"; street_number$
              CASE 4%
                 INPUT "STREET"; street$
              CASE 5%
                 INPUT "CITY"; city$
              CASE 6%
                 INPUT "STATE"; state$
              CASE 7%
                 INPUT "ZIP CODE"; zip$
              CASE 8%
                 INPUT "WAGE CLASS"; wage_class$
              CASE 9%
                 INPUT "DATE OF REVIEW"; date_of_review$
```

```
        CASE 10%
           INPUT "SALARY YTD"; salary_ytd$
        CASE 11%
           INPUT "TAX YTD"; tax_ytd$
        CASE ELSE
           PRINT "Invalid choice"
     END SELECT
  NEXT
END
```

Output

```
Changes? (please type YES or NO)? YES
1.  FIRST NAME
2.  LAST NAME
3.  STREET NUMBER
4.  STREET
5.  CITY
6.  STATE
7.  ZIP
8.  WAGE CLASS
9.  DATE OF REVIEW
10.  SALARY YTD
11.  TAX YTD

CHANGE NUMBER? 10
SALARY YTD? 14,277.08
Changes? (please type YES or NO)? YES
CHANGE NUMBER? 11
TAX YTD? 998.32
Changes? (please type YES or NO)? NO
```

See Chapter 7 and the *BASIC–PLUS–2 Reference Manual* for more information on the MAP statement.

10
Arrays

An array is a set of data that is ordered in any number of dimensions. This chapter describes how to create and use BASIC–PLUS–2 arrays.

10.1 Introduction

A one-dimensional array is called a *list* or *vector*. A two-dimensional array is called a *matrix*. BASIC–PLUS–2 arrays can have up to 8 dimensions, and they can be redimensioned at run time. In addition, you can specify the data type of the values in an array by using data type keywords or suffixes.

The subscript of an element in an array defines that element's position in the array. When you create an array, you specify the following:

- The number of dimensions that the array contains
- The range of values for the subscripts in each dimension of the array

BASIC–PLUS–2 arrays are zero-based; that is, when calculating the number of elements in a dimension, you count from zero to the number of elements specified. For example, an array with an upper bound of 10 has 11 elements: 0 through 10, inclusive. The array *My_array*(3,3) has 16 elements: 0 through 3 in each dimension, or 4^2.

To refer to an element in the array *Sales_data*, you need only specify the month you are interested in. For example, to print the information for the month of May, you would type the following:

```
10    DECLARE WORD CONSTANT JAN% = 1%
        .
        .
        .
      DIM Sales_month (12%)
      PRINT Sales_month (MAY%)     !Sales for month of May
```

You can create arrays either implicitly or explicitly. You implicitly create arrays having any number of dimensions by referencing an element of the array. If you implicitly create an array, BASIC–PLUS–2 sets the upper bound to 10 and the lower bound to zero. Therefore, any array that you create implicitly contains 11 elements in each dimension.

The following example refers to the array *Student_grades$*, thereby causing BASIC–PLUS–2 to create a one-dimensional array with that name.

```
10    Student_grades$(8) = "B"
```

You create arrays explicitly by declaring them in a DIM, DECLARE, COMMON, or MAP statement.

When you declare an array explicitly, the value that you give for the upper bound determines the maximum subscript value in that dimension. The upper bound must be a positive value.

You can use MAT statements to create and manipulate arrays. However, MAT statements are valid only on arrays of one or two dimensions.

10.2 Creating Arrays Explicitly

You can create arrays explicitly with four BASIC–PLUS–2 statements:

- DECLARE
- DIMENSION
- COMMON
- MAP

Normally, you can use the DECLARE statement to create arrays. However, in certain cases, you may want to create the array with another BASIC–PLUS–2 statement:

- You use the DIM statement to create virtual arrays and arrays that can be redimensioned at run time.
- You use the COMMON statement to create arrays that can be shared among program modules or to create arrays of fixed-length strings.
- You use the MAP statement to create an array and associate it with a record buffer, or to overlay the storage for an array, thus accessing the same storage in different ways.

When you create an array, the bounds you specify determine the array's size. The maximum value allowed for a bound can be as large as 32767; however, this number is actually limited by the amount of storage available to you. Very large arrays and arrays with many dimensions can cause fatal errors at both compile time and run time.

The following restrictions apply to arrays:

- When referencing an array, you must use the same number of subscripts as was specified in the DIM statement.
- You can use identical names for a simple variable and an array; for example, A% and A%(5,5). However, this is not recommended programming practice. If you use identical names for arrays with a different number of subscripts, for example, A(5) and A(10,10), BASIC–PLUS–2 prints the warning error "Inconsistent subscript usage" at compile time.
- If subscript checking is enabled, BASIC–PLUS–2 signals the error "Subscript out of range" (ERR=55) if you reference an array element whose subscripts are greater than the upper bound specified in the last execution of the DIM statement or less than zero.

The following sections explain how to declare arrays.

10.2.1 Creating Arrays with the DECLARE Statement

The DECLARE statement creates and names variables and arrays. All elements of arrays created with the DECLARE statement are initialized to zero or the null string. The following statement creates a LONGWORD integer array with 11 elements. Each element has an initial value of zero.

```
10    DECLARE LONG Ray(10)
```

Although BASIC–PLUS–2 initializes array elements to zero or the null string, it is good programming practice to initialize all array elements by assigning them values in your program. For example, the following program creates a three-dimensional string array and initializes each element to the null string:

```
100     DECLARE STRING second_array(10,10,10)
200     DECLARE LONG loop_1, loop_2, loop_3
300     FOR loop_1 = 0% TO 10%
                FOR loop_2 = 0% TO 10%
                        FOR loop_3 = 0% TO 10%
                                second_array(loop_1, loop_2, loop_3) = ""
                        NEXT loop_3
                NEXT loop_2
        NEXT loop_1
```

Note that the STRING data type with the DECLARE statement causes the creation of an array of dynamic strings. To create an array of fixed-length strings, declare the array in a COMMON or MAP statement.

10.2.2 Creating Arrays with the DIM Statement

The DIM statement creates and names one or more arrays. You should use the DIM statement to create an array only when you want to do the following:

- Redimension the array at run time
- Create a virtual array

When creating arrays with DIM, you specify the data type of the array elements with a data type keyword, a special suffix on the array name, or both. The array name can be any valid variable name. If you do not supply a data type keyword, the data type is determined by the suffix of the array name as follows:

- If the array name ends in a dollar sign, the array stores string data.
- If the array name ends in a percent sign, the array stores integer data.
- If the array name does not end in either a percent sign or a dollar sign, the array stores data of the default type. The default type is single-precision floating-point unless you change the default. See Chapter 4 for more information on default data types.

Even if the DIM statement contains a data type keyword, the array name can still end in the appropriate data type suffix. This makes the data type of the array immediately obvious.

The DIM statement can be either executable or declarative. If the specified bounds are constants, the DIM statement is declarative. This means that the storage is allocated at compile time, and the array cannot appear in any other DIM statement.

However, if any of the specified bounds are variables (simple or subscripted), the DIM statement is executable. This means that the storage for the array is allocated at run time, and the array can be redimensioned with a DIM statement any number of times.

Note

In the DIM statement, bounds can be either constants or variables (simple or subscripted), but not expressions.

When an array is redimensioned with the executable DIM statement, the array can become larger or smaller than it was. However, redimensioning an array in this way causes it to be reinitialized, and all data in the array is lost.

In contrast, MAT statements let you redimension an array to be the same size or smaller than it was. However, MAT statements redimension arrays only when assigning values or performing matrix I/O; therefore, the fact that MAT reinitializes the array does not matter. See Section 10.3.2 for more information on MAT statements.

10.2.2.1 Declarative DIM Statements

Declarative DIM statements are those with integer constants as bounds. The percent sign is optional for bounds; however, BASIC–PLUS–2 signals "Integer constant required" if a constant bound contains a decimal point. The following statement creates a 101-element virtual array containing string data. The elements of this array can each have a maximum length of 256 characters.

```
10    DIM #1%, STRING VIRT_ARRAY(100) = 256%
```

The following restrictions apply to the use of declarative DIM statements:

- A declarative DIM statement must lexically precede any reference to the array it dimensions.
- For declarative DIM statements, if you reference an array element whose subscripts are larger than the subscripts specified in the DIM statement, BASIC–PLUS–2 signals the error "Subscript out of range" (ERR=55).
- Because a declarative DIM statement allocates storage at compile time, an array of this type cannot appear in any other declarative statement such as a MAP, COMMON, DECLARE, or a later DIM statement.

10.2.2.2 Executable DIM Statements

Executable DIM statements are those with at least one variable bound. Bounds can be constants or simple variables, but at least one bound must be a variable. Executable DIM statements let you redimension an array at run time. The bounds of the array can become larger or smaller, but the number of dimensions cannot change. For example, you cannot redimension a four-dimensional array to be five-dimensional.

The executable DIM statement cannot be used on arrays in COMMON, MAP, DECLARE or declarative DIM statements, nor on virtual arrays or arrays received as formal parameters.

Whenever an executable DIM statement executes, it reinitializes the array. If you change the values of an executable DIM statement, the initial values are reset each time a DIM statement is executed.

In the following example, the second DIM statement reinitializes the array *real_array*; therefore, *real_array(1%)* equals zero in the second PRINT statement.

Example

```
10    X% = 10%
      Y% = 20%
      DIM real_array(X%)
      real_array(1%) = 100
      PRINT real_array(1%)
      DIM real_array(Y%)
      PRINT real_array(1%)
      END
```

Output

```
100
0
```

You cannot reference an array named in an executable DIM statement until after the DIM statement executes. If you reference an array element declared in an executable DIM statement whose subscripts are larger than the bounds specified in the last execution of the DIM statement, BASIC–PLUS–2 signals the error "Subscript out of range" (ERR = 55).

10.2.3 Creating Arrays with the COMMON Statement

You should create arrays with the COMMON statement when you need an array of fixed-length strings, or when you want to share an array among program modules. Program modules can share arrays in COMMON statements by defining a common block with the same name.

The COMMON statements in the following programs create a 100-element array of fixed-length strings, each element 10 characters long. Because the main program and subprograms use the same common name, the storage for these arrays is overlaid when the programs are linked. Therefore, both programs can read and write data to the array.

Example

```
10    !Main Program
      COMMON (ABC) STRING access_list(99) = 10

20    !Subprogram
      SUB SUB1
      COMMON (ABC) STRING new_list(99) = 10
```

10.2.4 Creating Arrays with the MAP Statement

You should create arrays with the MAP statement only when you want the array to be part of a record buffer, or when you want to overlay the storage containing the array. Note that string arrays in maps are always fixed-length.

You associate the array with a record buffer by naming the map in the MAP clause of the OPEN statement.

In the following example, the MAP statement creates two arrays: an 11-element fixed-length string array named *team* and a 33-element array of WORD integers named *bowling_scores*. Because the OPEN statement specifies MAP *ABC*, the storage for these arrays is used as the record buffer for the open file.

Example

```
10    MAP (ABC) STRING team(10) = 20, WORD bowling_scores(32)
      OPEN "BOWL.DAT" AS FILE #1%, SEQUENTIAL VARIABLE, MAP ABC
```

10.3 Creating Arrays Implicitly

There are two ways to create implicit arrays:

- By referencing an element of an array that has not been explicitly declared
- By using MAT statements

When BASIC–PLUS–2 first creates an implicit array, the lower bound is zero and the upper bound is 10. An array created by referencing an element can have up to eight dimensions. An array created with a MAT statement can have only one or two dimensions.

Note

The ability to create arrays implicitly exists for compatibility with previous implementations of BASIC. However, it is better programming practice to declare all arrays explicitly before using them.

10.3.1 Referencing an Undeclared Array Element

If you reference an element of an array that has not been explicitly declared, BASIC–PLUS–2 creates a new array with the name you specify. Arrays created by reference have default subscripts of (10), (10,10), (10,10,10) and so on, depending on the number of dimensions specified in the array reference. For example, the following program implicitly creates three arrays and assigns a value to one element of each.

Example

```
10    LET A(5,5,5) = 3.14159
      LET B%(3) = 33
      LET C$(2,2) = "Russell Scott"
      END
```

The first LET statement creates an 11 by 11 by 11 array that stores floating-point numbers and assigns the value 3.14159 to element (5,5,5). The second LET statement creates an 11-element list that stores integers and assigns the value 33 to element (3) and the third LET statement creates an 11 by 11 string array and assigns the value "Russell Scott" to element (2,2).

When you create an implicit numeric array by referring to an element, BASIC–PLUS–2 initializes all elements (except the one assigned a value) to zero. For implicit string arrays, BASIC–PLUS–2 initializes all elements (except the one assigned a value) to a null string. When you implicitly create an array, you cannot specify a subscript greater than 10. An attempt to do so causes BASIC–PLUS–2 to signal the error "Subscript out of range" (ERR = 55).

Note that you cannot create an array implicitly, then redimension the array with an executable DIM statement. The DIM statement must execute before any reference to the array.

An implicit array cannot appear in a declarative DIM statement. In the following example, the array is dimensioned *before* it is referenced, thus making it an explicitly declared array.

Example

```
10    DIM new_array(15,10,5)
      new_array(5,5,5) = 1
```

When referencing an array or function, be careful to specify the array name accurately or you will implicitly declare a new array. Unintentionally creating an array in your program can cause unexpected results. For example, when the following program lines are executed, the program returns the value in the array GETIT rather than invoking the GETINT function.:

Example

```
10    EXTERNAL WORD FUNCTION GETINT (WORD,WORD,WORD)
20    PRINT GETIT (2,7,3)
```

You can avoid this common error by using the OPTION TYPE = EXPLICIT statement in your programs.

10.3.2 Using MAT Statements

MAT statements let you assign values to or display entire arrays with a single statement. They also do the following:

- Implicitly create arrays
- Assign names to arrays
- Specify array dimensions
- Redimension existing arrays (to equal or smaller sizes)
- Assign element values
- Print the contents of arrays
- Perform matrix arithmetic

MAT statements are valid only on arrays of one or two dimensions. When MAT statements execute, they use row and column zero to store intermediate calculations. This means that MAT statements can overwrite data stored in row and column zero of your arrays, and you should not depend on data in these elements if your program uses MAT statements.

The default subscripts for arrays created implicitly with MAT statements are (10) or (10,10). The default is two dimensions. This means that if you create an array with a MAT statement and do not specify any subscripts, BASIC–PLUS–2 creates a two-dimensional, 11 by 11 array. If you specify a single subscript, BASIC–PLUS–2 creates a one-dimensional array with 11 elements.

Table 10–1 lists MAT statements and explains their functions.

Table 10–1 MAT Statements

Statement	Function
MAT	Assigns values of zero, 1, or a null string to array elements; also copies the values of one array to another and performs matrix arithmetic
MAT INPUT [#]	Assigns values to array elements from your terminal or a terminal-format file

(continued on next page)

Table 10–1 (Cont.) MAT Statements

Statement	Function
MAT LINPUT [#]	Assigns string values to string array elements from your terminal or from a terminal-format file
MAT PRINT [#]	Displays the contents of an array on your terminal, or writes array element values to a terminal-format file
MAT READ	Assigns DATA statement values to array elements

In the following example, the first MAT statement creates the string array *z_array$* with eight rows and eight columns and assigns a null string to all elements. The second MAT statement redimensions the array to six rows and six columns. The third MAT statement adds the values in each corresponding element of arrays *B* and *C* and stores the values in the corresponding elements of array *A*.

Example

```
10    MAT z_array$ = NUL$(7,7)
      MAT z_array$ = NUL$(5,5)
      MAT A = B + C
      END
```

10.3.2.1 The MAT Statement

The MAT statement can create an array and optionally assign values to all elements in that array. By specifying one of the MAT statement keywords, you can initialize arrays in one of four ways. Table 10–2 lists the MAT statement keywords and their functions.

Table 10–2 MAT Statement Keywords

MAT Keyword	Function
ZER	Sets the value of all elements in a numeric array to zero
CON	Sets the value of all elements in a numeric array to 1, except those in row and column zero
IDN	Sets the array to the identity matrix, that is, it sets the value of all elements in real or integer arrays to zero, except for those elements on the diagonal from element (1,1) to element (n,n), where n is the largest subscript in the array; the elements on the diagonal are set to 1 (IDN applies to square arrays only)
NUL$	Sets the value of all elements in a string array to the null string, except those in row and column zero

The array name can specify an existing array. MAT statements do not assign values to row and column zero.

Note that the MAT statement does not require subscripts. In the case of existing arrays:

- If you do not specify subscripts, BASIC–PLUS–2 does not change the current subscripts.
- If you specify subscripts, BASIC–PLUS–2 redimensions the array to the specified subscripts. When redimensioning arrays with the MAT statement, you cannot increase the total number of array elements (including those in row and column zero).

If you do not supply subscripts when creating an array with the MAT statement, BASIC–PLUS–2 assigns two subscripts, each with a value of 10. If you do specify subscripts, they define the dimensions of the array being implicitly created. Subscript values cannot exceed 10.

Example

```
10     DIM A(10,10), B(15), C(20,20)
20     MAT A = ZER                !Sets all elements of A to 0
30     MAT B = CON(10)            !Sets elements of B to 1; redimensions B
40     MAT C = IDN(10,10)         !Redimensions C to 10x10 identity matrix
50     PRINT "ARRAY A:"
       MAT PRINT A;
       PRINT
       PRINT "ARRAY B:"
60     MAT PRINT B;
       PRINT
       PRINT "ARRAY C:"
70     MAT PRINT C;
```

Output

```
ARRAY A:
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0

ARRAY B:
1  1  1  1  1  1  1  1  1  1
```

```
ARRAY 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
```

10.3.2.2 The MAT READ Statement

The MAT READ statement assigns values from DATA statements to array elements. Subscripts define either the dimensions of the array being created or the new dimensions of an existing array; subscripts are optional in MAT READ statements.

If you do not provide enough data in DATA statements to fill the specified array, BASIC–PLUS–2 leaves the remaining array elements unchanged. If you provide more data values than there are array elements, BASIC–PLUS–2 assigns enough values to fill the array and leaves the DATA pointer at the next value.

In the following example, BASIC–PLUS–2 fills matrix *B* with the first four DATA items, fills matrix *C* with the next four DATA values, and leaves the DATA pointer at the ninth value in the DATA list.

Example

```
10     MAT READ B(2,2)
       MAT READ C(2,2)
       PRINT
       PRINT "MATRIX B"
       PRINT
       PRINT
       MAT PRINT B;
       PRINT
20     PRINT "MATRIX C"
       PRINT
       PRINT
       MAT PRINT C;
       DATA 1,2,3,4,5,6,7,8,9,10
30     END
```

Output

```
MATRIX B

 1  2
 3  4
MATRIX C

 5  6
 7  8
```

10.3.2.3 The MAT INPUT [#] Statement

The MAT INPUT statement assigns values from your terminal to array elements. The MAT INPUT # statement reads data from a terminal-format file and writes it to an array. The optional subscripts in a MAT INPUT statement define either the dimensions of the array being created implicitly or the new dimensions of an existing array. If you are implicitly creating the array, the value of a subscript cannot exceed 10.

The MAT INPUT statement requests data from your terminal, as does the INPUT statement; it prints a question mark (?) prompt that you can disable with the SET NO PROMPT statement and then enable with the SET PROMPT statement. However, you cannot include a string prompt with the MAT INPUT statement.

When you enter a series of values separated by commas, BASIC–PLUS–2 enters the values you supply into successive array elements by row, starting with element (1,1) and filling row 1 before starting row 2. If you provide fewer data items than there are elements, the remaining elements are unchanged. If you provide more items than there are elements, BASIC–PLUS–2 ignores the excess.

The MAT INPUT # statement takes values from an open file and assigns them to the matrix elements by rows, starting with element (1,1). It fills the elements in row 1 before starting row 2. The file can have one or more values in each record; however, multiple values must be separated with commas.

In the following example, the open file on channel 3 contains the following data: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13. The MAT INPUT # statement reads this data and uses it to fill array *A*, filling in row 1 before beginning row 2. The MAT INPUT B(2,2) statement dimensions array *B* to 9 elements (0 to 2 in each dimension) and provides values for all the elements except those in row and column zero.

Example

```
10    MAT INPUT #3, A
      PRINT
      MAT PRINT A;
      MAT INPUT B(2,2)
      PRINT
      MAT PRINT B;
```

Output

```
  1  2  3  4  5  6  7  8  9  10
 11 12 13  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0

? 1,2,3,4

 1  2
 3  4
```

Note that the MAT PRINT statement does not print row and column zero. For more information on the MAT PRINT statement, see Section 10.3.2.5.

The MAT INPUT statement can also redimension an existing array.

Example

```
10    DIM new_array%(5,5)
      MAT INPUT new_array%(2,4)
      MAT PRINT new_array%;
      END
```

Output

```
? 1,2,3,4,5,6,7,8

1 2 3 4
5 6 7 8
```

When entering values in response to MAT INPUT, you can enter an ampersand as the last character on the line and continue on the next line.

10.3.2.4 The MAT LINPUT [#] Statement

The MAT LINPUT statement assigns string values to string array elements. The MAT LINPUT # statement reads string values from a terminal-format file and writes them to a string array.

The MAT LINPUT statement prompts for individual array elements. It fills the array by rows, starting with element (1,1). It assigns the line you supply (including commas, semicolons, and quotation marks, but excluding the line terminator) to an array element.

Example

```
10    DIM emp_nam$(5,5)
      MAT LINPUT emp_nam$(2,2)
      PRINT emp_nam$(1,1)
      PRINT emp_nam$(1,2)
      PRINT emp_nam$(2,1)
      PRINT emp_nam$(2,2)
      END
```

Output

```
? HODGES
? LAFFERTY
? ELDON
? HOPKINS
HODGES
LAFFERTY
ELDON
HOPKINS
```

By specifying the subscripts (2,2), MAT LINPUT redimensions the array to four elements and overwrites the old values. BASIC–PLUS–2 then prompts for these elements.

MAT LINPUT # also excludes line terminators when assigning values to string array elements. MAT LINPUT # places the values from the open file into the specified array, filling the array by rows, starting with element (1,1). If there are more values in the file than there are array elements, BASIC–PLUS–2 ignores the excess. If there are fewer, BASIC–PLUS–2 assigns a null string to the remaining elements.

The following program reads 50 records from the open disk file and assigns them to the array named *part_name$*. If there are more than 50 records in the file, BASIC–PLUS–2 ignores the excess. If there are fewer than 50 records, then BASIC–PLUS–2 fills the remaining elements of the array with the null string.

Example

```
10     DIM part_name$(50)
       MAT LINPUT #1%, part_name$
```

10.3.2.5 The MAT PRINT [#] Statement

The MAT PRINT statement prints some or all of an array's elements, excluding row and column zero. The MAT PRINT # statement takes values from an array by row, starting with element (1,1), and writes each element to a sequential record in the terminal-format file.

Subscripts are optional in MAT PRINT statements. If you do not specify subscripts, MAT PRINT displays the entire array, excluding row and column zero. If you specify subscripts, MAT PRINT displays the specified subset of the array. In the case of the MAT PRINT # statement, the subscripts determine how many array elements are written to the file. The MAT PRINT [#] statement does not redimension an existing array.

If the last character in the MAT PRINT [#] array list is a semicolon, BASIC–PLUS–2 begins each array row on a separate line. Data values on each line are packed together with no intermediate spaces. However, if the last character in the MAT PRINT [#] arrays list is a comma, BASIC–PLUS–2 begins each array row on a separate line and each data value in a separate print zone.

If there is neither a comma nor a semicolon after the array name, BASIC–PLUS–2 prints each array element on a separate line. In the following example, the first MAT PRINT statement does not end in a comma or semicolon, so each element is printed on a separate line. The second MAT PRINT statement prints the elements twice, the first time starting each element in a new print zone, and the second time leaving a space before and after each value. The MAT PRINT # statement sends the last two lines of output to a terminal-format file.

Example

```
10     MAT INPUT A(5)
       PRINT
       MAT PRINT A
       PRINT
       MAT PRINT A, A;
       MAT PRINT #3, A, A;
       END
```

Output

```
? 5

 5
 0
 0
 0
 0

 5             0             0             0             0

 5  0  0  0  0
```

10.3.2.6 Matrix I/O Functions

MAT statements do not signal error messages when there are more data items than array elements to contain them, or when there are fewer data items than array elements to contain them.

BASIC–PLUS–2 provides two functions that let you determine how much data the MAT statements transfer: NUM and NUM2.

For two-dimensional arrays, the NUM function returns an integer value specifying the row number of the last data item transferred, whereas the NUM2 function returns an integer value specifying the column number of the last data item transferred. For one-dimensional arrays, the NUM function returns the number of items entered, whereas the NUM2 function returns a zero.

With these functions, you can determine the number of items transferred from a terminal-format file. Note, however, that you cannot use the NUM and NUM2 functions to implicitly declare an array. In the following example, the terminal-format file EMP.DAT contains the values 1 through 17, inclusive. When these values are read using the MAT INPUT # statement, NUM and NUM2 represent the row and column number, respectively, of the last value read.

Example

```
10     OPEN "EMP.DAT" FOR INPUT AS FILE #3%
       DIM emp_name$(5,5)
       MAT INPUT #3%, emp_name$
20     PRINT NUM, NUM2
30     END
```

Output

```
4                2
```

10.4 Array Input and Output

You can assign values to array elements from within your program, from an external source, such as terminal input or from files, or with MAT statements.

You can write data from an array with the following statements:

- PRINT
- MAT PRINT
- MAT PRINT #

The following sections tell you how to perform input and output operations on BASIC–PLUS–2 arrays.

10.4.1 Assigning Values with the LET Statement

The LET statement assigns values to individual array elements.

Example

```
10     DIM voucher_num%(100)
   .
   .
   .
60     LET voucher_num%(20) = 3253%
   .
   .
   .
100    END
```

You can also assign values to a portion of an array with the LET statement and a FOR...NEXT loop. In the following example, the FOR...NEXT loop assigns zero to array elements (1,5) through (1,10), (2,5) through (2,10), and (3,5) through (3,10).

Example

```
10     DIM po_number%(100,100)
   .
   .
   .
       FOR I% = 1% TO 3%
          FOR J% = 5% TO 10%
              LET po_number%(I%,J%) = 0%
          NEXT J%
       NEXT I%
   .
   .
   .
       END
```

10.4.2 Listing Array Elements with the PRINT Statement

You print individual array elements by naming those elements in the PRINT statement. For example:

```
10    PRINT parts_list$(35%)
```

With a FOR...NEXT loop, you can print all or part of an array.

Example

```
10    DIM capture_ratio(10,10)
    .
    .
    .
      FOR Y% = 7% TO 10%
         FOR X% = 7% TO 10%
            PRINT capture_ratio(X%,Y%)
         NEXT X%
      NEXT Y%
```

10.5 Matrix Operators

BASIC–PLUS–2 provides a special set of MAT statements for array computations. These statements enable you to add, subtract, and multiply matrices, and to assign values to elements. Note that if you specify an array without subscripts (for example, MAT A), the default is two dimensions.

BASIC–PLUS–2 also provides matrix functions to transpose and invert matrices, and to find the determinant of a matrix you invert.

Note

MAT operators do not operate on elements in row or column zero.

10.5.1 Arithmetic Matrix Operations

MAT operators perform matrix assignment, addition, subtraction, and multiplication.

All of these operations use the keyword MAT, followed by an expression. If the array has not been previously dimensioned, these operations create an array. The created output array's dimensions depend on the operation performed, but must be (10,10) or smaller.

Note

You can use the MAT operators on arrays larger than (10,10) if the input and output arrays are explicitly created or received as a formal parameter.

10.5.1.1 Assignment

You can assign all values in one array to another array with the MAT statement. In the following example, each element of *new_array* is set to the corresponding element in *old_array*. The dimensions of *new_array* are also redimensioned to the dimensions of *old_array*.

```
10    MAT new_array = old_array
```

10.5.1.2 Addition and Subtraction

You can add the elements of two arrays. In the following statement, the two input lists, *first_list%* and *second_list%*, must have identical dimensions. The elements of the new list, *sum_list%*, equal the sum of the corresponding elements in the input lists.

```
10    MAT sum_list% = first_list% + second_list%
```

You can also subtract the elements of two arrays. The following program subtracts one array from another.

Example

```
10    DIM first_array(30,30)
      DIM second_array(30,30)
      DIM difference_array(30,30)
  .
  .
  .
      MAT difference_array = first_array - second_array
```

Each element of *difference_array* is the arithmetic difference of the corresponding elements of the input arrays.

10.5.1.3 Multiplication

You can multiply the elements of two arrays, provided that the number of columns in the first array equals the number of rows in the second array. The resulting array contains the product of the two input arrays.

Example

```
10    DIM A(2,2), B(2,2), C(2,2)
      A(1,1) = 1
      A(1,2) = 2
      A(2,1) = 3
      A(2,2) = 4
      B(1,1) = 5
      B(1,2) = 6
      B(2,1) = 7
      B(2,2) = 8
20    MAT C = A * B
      MAT PRINT C
```

Output

```
19
22
43
50
```

You can also multiply a matrix by a scalar quantity. BASIC–PLUS–2 multiplies each element of the input array by the scalar quantity you supply. The output array has the same dimensions as the input array. Enclose the scalar quantity in parentheses. The following example multiplies the elements of *inch_array* by the inch-to-centimeter conversion factor and places these values in *cm_array*.

Example

```
10    DIM inch_array(5), cm_array(5)
      MAT READ inch_array
      DATA 1,12,36,100,39.37
      MAT cm_array = (2.54) * inch_array
      MAT PRINT cm_array,
      END
```

Output

```
2.54     30.48     91.44     254     99.9998
```

10.5.2 Matrix Functions

BASIC–PLUS–2 provides three matrix functions:

- TRN
- INV
- DET

With these functions, you can transpose and invert matrices, and find the determinant of an inverted matrix.

10.5.2.1 The TRN Function

The TRN function transposes a matrix. When you transpose a matrix, BASIC–PLUS–2 interchanges the array's dimensions. For example, a matrix with *n* rows and *m* columns is transposed to a matrix with *m* rows and *n* columns. The elements in the first row of the input matrix become the elements in the first column of the output matrix. You cannot transpose a matrix to itself; MAT A = TRN(A) is invalid.

This example creates a 3 by 5 matrix, transposes it, and prints the results.

Example

```
10    DIM B(3,5)
      MAT READ B
      MAT A = TRN(B)
      DATA 1,2,3,4,5
      DATA 6,7,8,9,10
      DATA 11,12,13,14,15
      MAT PRINT B;
      MAT PRINT A;
      END
```

Output

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15

1  6   11
2  7   12
3  8   13
4  9   14
5  10  15
```

10.5.2.2 The INV Function

The INV function inverts a matrix. BASIC–PLUS–2 can invert a matrix only if its subscripts are identical and it can be reduced to the identity matrix by elementary row operations. The input matrix multiplied by the output matrix (its inverse) always gives the identity matrix as a result.

Example

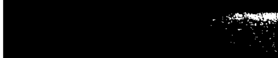
```
10    MAT INPUT first_array(3,3)
      MAT PRINT first_array;
      PRINT
      MAT inv_array = INV (first_array)
      MAT PRINT inv_array;
      PRINT
      MAT mult_array = first_array * inv_array
      MAT PRINT mult_array;
```

Output

```
? 4,0,0,0,0,2,0,8,0
 4  0  0
 0  0  2
 0  8  0

 .25  0  0
 0  0  .125
 0  .5  0

 1  0  0
 0  1  0
 0  0  1
```

10.5.2.3 The DET Function

The DET function returns the determinant of a matrix. The DET function returns a floating-point number that is the determinant of the last matrix inverted. If you use the DET function before inverting a matrix, the value of DET is zero.



11
Program Segmentation

Program segmentation is the process of writing a program as a group of small, manageable routines and modules, rather than as a large single module. This chapter describes how to do the following:

- Create and invoke BASIC–PLUS–2 subprograms
- Share data among program modules
- Build a single task image from multiple program modules
- Call non-BASIC–PLUS–2 subprograms

11.1 Introduction

Writing large programs as a group of program modules provides several advantages for the programmer. For instance, if you write a large program as a single module, the program is hard to understand and debug; however, if you program a group of program modules so that each module performs a single logical function of the task, you get the following advantages:

- Faster program design and implementation
- Faster debugging and testing
- Program code that is easier to understand
- Program code that is easier to maintain
- Program code that is easier to transport

Also, a large single module may require more address space than the 32K word limit. When you write the program as a group of program modules, each module can be called into memory as needed, thus reducing a program's memory requirements.

A program module is a block of code that is created and compiled separately and then invoked or called from another program module. Program modules can be either *main programs* or *subprograms*.

A single executable task contains only one main program, but can contain several subprograms. The main program is the program module that begins the execution of the task. The main program then calls the first subprogram, which can in turn call another subprogram, and so on. After each subprogram completes execution, control returns to the program module that called it.

Note that subprogram invocations can be nested. That is, a subprogram called from a main program can call another subprogram; however, a subprogram cannot call itself or the subprogram that called it.

11.2 BASIC–PLUS–2 Subprograms

There are two types of BASIC–PLUS–2 subprograms:

- SUB subprograms
- FUNCTION subprograms

You use the CALL statement to invoke a SUB subprogram. You invoke a FUNCTION subprogram just as you would a built-in BASIC–PLUS–2 function.

SUB and FUNCTION subprograms are similar to GOSUB and DEF function subroutines. For example, CALL and GOSUB statements both transfer control to another part of an executable image. After processing is complete, END SUB and RETURN statements both return control to the statement following the CALL or GOSUB statement that transferred control. Similarly, DEF and FUNCTION statements both let you define functions that return values when invoked.

The difference between subroutines and subprograms lies in the interface between the parts of the program. When you divide a large program into small program modules, the interface between the modules is strictly defined and well controlled. Variables and data can be shared between program modules only by way of parameters, COMMON or MAP statements, or files. Thus, it is much more difficult for one program module to cause unexpected side effects in another program module.

The interface between a DEF or GOSUB subroutine and the surrounding program is much less clearly defined; all program variables and data are accessible from within a DEF or GOSUB subroutine. Thus, it is much easier for a DEF or GOSUB to alter or redefine variables unintentionally.

This is not to say that you should never use DEF or GOSUB subroutines; however, you should be aware of the possibility of side effects and be more careful in coding them. Considering possible side effects is even more important if a program may be modified by other programmers. The following sections describe SUB and FUNCTION subprograms.

11.2.1 SUB Subprograms

The SUB statement must be the first statement on the first numbered line of a BASIC–PLUS–2 SUB subprogram. The format of the SUB statement is as follows:

SUB sub-name ([[data-type] param]), . . .

sub-name
Is a unique, 1- to 6-character name. Neither the subprogram nor main program can have a map or a common block of this name.

data-type
Is a data-type keyword. See Chapter 7 for more information about data-type keywords.

param
Is a parameter. You can specify up to 32 parameters.

Note

For both SUB and FUNCTION subprograms, the Task Builder requires a name containing only members of the RAD-50 character set. See C for information on the RAD-50 character set. The subprogram name can begin with a dollar sign ($) only if the name is enclosed in quotes. It is recommended that you do not begin subprogram names with a dollar sign, as names of this form are reserved for Digital.

The parameters in the SUB statement must agree in number and data type with the parameters in the calling statement.

The subprogram name can be either a quoted or an unquoted string. For example, these are valid subprogram names:

```
10 SUB  "SUBPRG"
10 SUB  'SUBPRG'
10 SUB  SUBPRG
```

You can include a dollar sign ($) or a period (.) in a subprogram name; however, if a dollar sign or a period is the first character in the name, the entire name must be enclosed in quotes.

The END SUB statement marks the end of a BASIC–PLUS–2 subprogram. The format of the END SUB statement is as follows:

END SUB

END SUB transfers control to the statement immediately after the statement that called the subprogram. The END SUB statement must be the last statement in the subprogram. The SUBEND statement is identical to the END SUB statement.

You exit from a subprogram with the EXIT SUB statement. The format of the EXIT SUB statement is as follows:

```
EXIT SUB
```

The EXIT SUB statement is equivalent to an unconditional transfer to the END SUB statement.

11.2.2 FUNCTION Subprograms

The FUNCTION statement marks the beginning of a FUNCTION subprogram. Its format is as follows:

```
FUNCTION data-type func-name ([ [data-type] param ], . . . )
```

data-type
Is a data-type keyword. The data-type keyword following FUNCTION specifies the data type of the return value. A data-type keyword preceding a parameter specifies the data type of that parameter. See Chapter 7 for more information about data-type keywords.

func-name
Is a unique, 1- to 6-character subprogram name.

param
Is a parameter. You can specify up to eight parameters.

The END FUNCTION statement marks the end of a function subprogram, returns a value to the calling program, and returns program control to the statement that called the function subprogram. The format of the END FUNCTION statement is as follows:

```
END FUNCTION [exp]
```

exp
Is the function result unless an EXIT FUNCTION statement is executed. The optional expression must be compatible with the DEF or FUNCTION data type. This expression supersedes all function assignments.

The EXIT FUNCTION statement returns program control to the statement that invoked the function subprogram. It is equivalent to an unconditional transfer to the END FUNCTION statement. The format of the EXIT FUNCTION statement is as follows:

EXIT FUNCTION [exp]

exp
Is the function result. It supersedes any function assignment. It also overrides any expression specified on the END DEF or END FUNCTION statement.

11.3 Declaring BASIC–PLUS–2 Subprograms

You declare the existence of a subprogram by naming it in an EXTERNAL statement. Declaring subprograms is optional for SUB subprograms; the CALL statement that invokes a SUB subprogram is an implicit declaration; however, you must delare FUNCTION subprograms with the EXTERNAL statement.

The EXTERNAL statement also lets you declare the data type for each parameter passed to the subprogram. If the subprogram is a FUNCTION subprogram, the EXTERNAL statement requires you to specify the data type of the returned value. To declare BASIC–PLUS–2 subprograms, the format for the EXTERNAL statement is:

EXTERNAL SUB {sub-name ([[data-type], . . .])}, . . .

EXTERNAL data-type FUNCTION {func-name ([[data-type], . . .])}, . . .

SUB
Specifies that the external procedure is a BASIC–PLUS–2 SUB subprogram.

sub-name
The 1- through 6-character name of the SUB subprogram.

FUNCTION
Specifies that the external procedure is a BASIC–PLUS–2 FUNCTION subprogram.

func-name
The 1- through 6-character name of the FUNCTION subprogram.

data-type
Is a data-type keyword. The data-type keyword preceding FUNCTION specifies the datatype of the value returned by the subprogram. A data-type

keyword within the parameter list specifies the datatype of that parameter. See Chapter 7 for more information about data-type keywords.

Data-type keywords in the parameter list apply to all the remaining parameters until you specify a new data type. If you do not specify a parameter's data type, the default is determined by either the compiler default, the OPTION statement, or a BASIC–PLUS–2 qualifier.

By declaring all subprograms before calling them, you gain several advantages:

- The program is easier to read and maintain.
- Data types need not be specified when you invoke the subprograms.
- BASIC–PLUS–2 automatically converts expressions to the proper data type when passing them as parameters.

11.4 Accessing BASIC–PLUS–2 Subprograms

You access a FUNCTION subprogram by declaring it in an EXTERNAL statement and by invoking the function name as you would a BASIC–PLUS–2 built-in function. You access a SUB subprogram with the CALL statement.

The CALL statement transfers control to a subprogram, and optionally passes arguments to it. The format of the CALL statement is as follows:

CALL sub-name ([param], . . .)

sub-name
Is a unique, 1- to 6-character name. Neither the subprogram nor main program can have a map or a common block of this name.

param
Represents 1 through 8 optional parameters that BASIC–PLUS–2 passes from the calling program to the subprogram. The parameters must agree in data type and number with the parameters you define in the SUB statement of the subprogram. You can also pass null parameters.

The subprogram name can be either a quoted or an unquoted string; however, you cannot use string variables to specify a subprogram name because BASIC–PLUS–2 interprets the string variable as the actual subprogram name. In the following example, BASIC–PLUS–2 attempts to name the subprogram NAM$:

```
10 NAM$ = "SUBPRG"
20 CALL NAM$
```

When BASIC–PLUS–2 accesses a subprogram, it transfers control from the calling program to the SUB or FUNCTION statement in the subprogram. It also passes the specified parameters to the subprogram. A simple example of a main program that calls a subprogram follows.

Main Program

```
100     EXTERNAL SUB SUB01(LONG, LONG, LONG)
        DECLARE LONG A, B, C
        INPUT "Please type three integers"; A, B, C
        CALL SUB01 (A, B, C)
32767   END
```

SUB Subprogram

```
100     SUB SUB01 (LONG X, LONG Y, LONG Z)
        PRINT "The sum is"; X + Y + Z
32767   END SUB
```

The main program prompts for three integers: A, B, and C. It then passes these variables as parameters to the SUB subprogram. The subprogram prints the sum of these variables and returns control to the calling program.

The following example performs the same task using a FUNCTION subprogram.

Main Program

```
100     EXTERNAL LONG FUNCTION FUN01(LONG, LONG, LONG)
        DECLARE LONG A, B, C
        INPUT "Please type three integers"; A, B, C
        PRINT "The sum is"; FUN01(A, B, C)
32767   END
```

FUNCTION Subprogram

```
100     FUNCTION LONG FUN01 (LONG X, LONG Y, LONG Z)
        FUN01 =  X + Y + Z
32767   END FUNCTION
```

These two sets of programs perform essentially the same operation; however, the SUB subprogram performs the addition and displays the sum, while the FUNCTION subprogram returns a value to the main program, and the main program prints the sum of the variables.

Note that when coding FUNCTION subprograms, you must specify a data type for the function in both the main program EXTERNAL statement and the subprogram FUNCTION statement. This data-type keyword specifies the data type of the value returned by the function subprogram.

11.5 Passing Parameters to a BASIC–PLUS–2 Subprogram

When you invoke a subprogram (with a CALL statement, for example), you can specify up to eight parameters to be passed to the subprogram. This parameter list contains *actual parameters*. The actual parameters specify the actual values used by the subprogram. When you declare a subprogram (for example, in an EXTERNAL statement), the parameter list contains *formal parameters*. The formal parameters specify the order and data type of the information passed to the subprogram.

Note that a formal parameter list appears in both the EXTERNAL and SUB or FUNCTION statements. You must make sure that these formal parameter lists match the number, order, and data types of the parameters. BASIC–PLUS–2 does not signal an error if the formal parameter lists do not match, because when you use the EXTERNAL statement, BASIC–PLUS–2 converts them for you.

Parameters can be any of the following:

- Constants
- Variables
- Expressions
- Function references
- Array elements
- Entire arrays

Parameters can be modifiable or nonmodifiable. If the parameter is modifiable, the value you assign to the parameter in the subprogram replaces the value in the calling program.

Modifiable parameters include:

- Entire arrays
- Simple string variables
- Simple numeric variables
- Common block or map elements

The term *simple* means unsubscripted.

If the parameter is nonmodifiable, the value you assign the parameter in the subprogram does not replace the value used by the calling program.

Nonmodifiable parameters include:

- Constants
- Expressions
- Function references
- Individual array elements

When a program passes a modifiable parameter to a subprogram, it actually passes a pointer to the parameter's address in memory. If the subprogram assigns a new value to the parameter, the new value replaces the original value in memory, and the calling program can then access the modified value.

If a parameter is an expression, a function reference, or an array element, it is not possible to pass the parameter's address. In these cases BASIC–PLUS–2 makes a local copy of the parameter's value and passes the address of the local copy. The subprogram can modify only the local copy and not the actual parameter.

You can force BASIC–PLUS–2 to make a local copy of any parameter by enclosing it in parentheses. In the following example, the calling program passes one modifiable and one nonmodifiable parameter to a subprogram.

Main Program

```
10      DECLARE WORD A, B(10,10)
        EXTERNAL SUB XYZ (WORD, WORD)
        A = 5
        B(5,5) = 10
        PRINT "A equals"; A
        PRINT "B(5,5) equals"; B(5,5)
        CALL XYZ ( A, B(5,5) )
        PRINT "A equals"; A
        PRINT "B(5,5) equals"; B(5,5)
32767   END
```

Subprogram

```
100     SUB XYZ (WORD AAA, WORD BBB)
        AAA = 100
        BBB = 500
        PRINT "AAA equals"; AAA
        PRINT "BBB equals"; BBB
32767   SUBEND
```

Output

```
A equals 5
B(5,5) equals 10
AAA equals 100
BBB equals 500
A equals 100
B(5,5) equals 10
```

In the main program:

1. The DECLARE statement declares a word integer named A and a 121-element array of word integers named B.
2. The EXTERNAL statement declares an external SUB subprogram named XYZ. The EXTERNAL statement has a formal parameter list specifying that the subprogram receives two word integers as parameters.
3. The next two statements assign a value of five to A and a value of 10 to the array element (5,5) in list B.
4. The PRINT statements display the values of A and B(5,5).
5. The CALL statement calls the subprogram XYZ and passes A and B(5,5) as actual parameters.
6. The next two PRINT statements display the values of A and B(5,5) after control has returned from the subprogram.
7. The END statement ends the main program.

In the subprogram:

1. The SUB statement identifies the program module as a SUB subprogram receiving two word integers as parameters. The formal parameter list names these parameters AAA and BBB.
2. The next two statements assign a value of 100 to AAA and 500 to BBB.
3. The PRINT statements display these new values.
4. The SUBEND statement ends the subprogram and returns control to the main program.

The subprogram assigns a new value to each parameter; however, when control returns to the main program, the first parameter has a new value while the second parameter does not. This is because a simple variable is modifiable while an individual element of an array is not.

Single array elements are nonmodifiable when passed to the subprogram as parameters in the CALL statement; however, if you pass an entire array as a parameter, you can modify one or all of the elements in that array.

To pass an entire array you must use a special format. In the CALL statement, you specify the array name followed by *n* commas enclosed in parentheses, where *n* is the number of array dimensions minus one. For example:

```
!For one-dimensional arrays:
CALL A50()
!For multi-dimensional arrays:
CALL D_ARRAY(,,,)
```

In the EXTERNAL statement you replace the array name with a data type specifying the array's data type and the DIM keyword indicating that the parameter is an array. For example:

```
DIM() or BYTE DIM(,,)
```

The following example passes an entire array to a subprogram.

Main Program

```
10      DECLARE STRING X(2,2)
        EXTERNAL SUB CCC ( STRING DIM(,) )
        MAT X = NUL$
        CALL CCC( X(,) )
        MAT PRINT X
32767   END
```

Subprogram

```
100     SUB CCC ( STRING ABC(,) )
        ABC(1,1) = "XYZ"
32767   SUBEND
```

Output

```
XYZ
```

In the main program:

1. The DECLARE statement declares a two-dimensional string array named X.
2. The EXTERNAL statement declares an external SUB subprogram named CCC. The parameter list specifies that the subprogram receives a two-dimensional string array as a parameter.
3. The MAT statement sets all elements of string array X to the null string.
4. The CALL statement calls the subprogram CCC and passes the string array X to it as a parameter.
5. The MAT PRINT statement displays the array after control has returned from the subprogram.
6. The END statement ends the main program.

In the subprogram:

1. The SUB statement identifies the program module as a SUB subprogram receiving one string array as a parameter. The formal parameter list names this parameter ABC.
2. The next statement assigns a value of "XYZ" to element (1,1) of the string array.
3. The SUBEND statement ends the subprogram and returns control to the calling program.

The output from this example shows that entire arrays are modifiable. The entire array X is passed as a parameter, and the value assigned in the subprogram remains when control returns to the calling program.

Passing entire virtual arrays as parameters in the CALL statement is not recommended. Instead, you can share the data in a virtual array between a calling program and a subprogram by opening a virtual file in either program and dimensioning the array (using the same channel number) in both programs.

You must dimension a virtual array before opening the corresponding virtual file. The two programs need not call the virtual array by the same name or use the same dimensions; however, using the same dimensions reduces the risk of error.

Note that any array redimensioned in a subprogram is redimensioned in the calling program as well.

11.6 Sharing Data Between Program Modules

In addition to passing parameters, there are three methods for sharing data between program modules:

- Common blocks
- Maps
- Files

Common blocks and maps are similar, but it is suggested that common blocks be used to share the data in program variables and arrays, whereas maps should be used only for sharing I/O record buffers between modules. Files should be used to share data between programs when accessing a large amount of data.

11.6.1 Common Blocks and Maps

COMMON and MAP statements let you share data among program modules. These statements define a named area of memory called a program section (PSECT) containing data that can be shared between any module that defines a PSECT with the same name.

In a single program module, two COMMON statements with the same name are concatenated. That is, the storage for the variables named in the second common block is appended to the storage for the variables named in the first common block; however, if the same common block name is used in both a calling program and a subprogram, the storage for the subprogram's common block overlays the storage for the calling program's common block. Thus, a calling program variable name and a subprogram variable name can point to the same storage location.

MAP statements with the same name always overlay the same storage, regardless of the module in which they reside. Calling programs and subprograms should share a map area only if they are performing I/O and need to share a record buffer.

Using a common block or map to share data rather than passing parameters in a CALL statement has the following advantages:

- BASIC–PLUS–2 can access the data more quickly.
- You can share a larger amount of data.

The COMMON statement has the following format:

```
{ COM    }
{ COMMON }[ (com-name) ] { [ data-type ] com-item }, . . .
```

com-name
Is a 1- to 6-character name you assign to the common block. Common blocks cannot have the same name as a program module within a single task image; however, a common block can have the same name as a map. This means that they define the same storage. If you do not specify a name, .$$$$. is the default name for the common block.

data-type
Is a data-type keyword. See Chapter 7 for information on data types.

com-item
Can be a variable, an array, or a FILL item. See Chapter 7 for more information on FILL items.

Define the common block or map area in your main program and include the same COMMON or MAP statement in your subprogram to access the data. For example:

Main Program

```
10      COMMON (RESERV) STRING STRI_NG = 44, REAL RE_AL
        STRI_NG = "Here is the value in the calling program:"
        RE_AL = 123
        PRINT STRI_NG;RE_AL
        CALL SUB1
        PRINT STRI_NG;RE_AL
32767   END
```

Subprogram

```
10      SUB SUB1
        COMMON (RESERV) STRING STRI_NG = 44, REAL RE_AL
        STRI_NG = "Here is the value after the call:"
        RE_AL = 345
32767   SUBEND
```

Output

```
Here is the value in the calling program: 123
Here is the value after the call: 345
```

The MAP statement has the following format:

MAP (map-name) { [data-type] map-item }, . . .

map-name
Is a 1- to 6-character name you assign to the map. Maps cannot have the same name as a subprogram within a single task image; however, a map can have the same name as a common block. This means that they define the same storage.

data-type
Is a data-type keyword. See Chapter 7 for information on data types.

map-item
Can be a variable, an array, or a FILL item. See Chapter 7 for more information on FILL items.

The following example shows a record buffer being shared by two program modules.

Main Program

```
100     MAP (BUF)  STRING PART.NAME = 20,      &
                   WORD   EXPIRE.DATE
        OPEN "FILE.DAT" AS FILE #3,            &
                   ORGANIZATION RELATIVE,      &
                   MAP BUF
        FOR REC.NUM% = 1% TO 50%
           GET #3, RECORD REC.NUM%
           CALL SUB1(REC.NUM%) IF EXPIRE.DATE = 365%
        NEXT REC.NUM%
        CLOSE #3
32767   END
```

Subprogram

```
100     SUB SUB1 ( INTEGER N )
        MAP (BUF) STRING PART.NAME = 20,       &
                  WORD   EXPIRE.DATE
        PRINT "Old part name", PART.NAME
        INPUT "New part name", PART.NAME
        EXPIRE.DATE = 1%
        PUT #3, RECORD N
32767   END SUB
```

The main program opens a file using MAP BUF as its record buffer. If the variable EXPIRE.DATE is equal to 365, the subprogram SUB1 is called. The subprogram defines MAP BUF, displays the current value of PART.NAME, modifies the record buffer, and writes the new record to the file.

In both common blocks and maps, simple numeric variables reserve the following:

- 1 byte of storage for BYTE integers
- 2 bytes of storage for WORD integers
- 4 bytes of storage for LONG integers
- 4 bytes of storage for single-precision, floating-point numbers
- 8 bytes of storage for double-precision, floating-point numbers

---

Note

Examples and explanations in this section assume single-precision, floating-point variables are used.

String variables reserve fixed amounts of storage. The default length is 16 bytes. You can reserve more or less space by defining lengths for the string variables in the MAP or COMMON statement.

```
10 COMMON (RESERV) STRING A = 10, STRING B, WORD C
```

In this example, BASIC–PLUS–2 reserves a total of 28 bytes for the common block RESERV, 10 bytes for A, 16 bytes for B (the default), and 2 bytes for C.

You can redefine the area of a common block or map between program modules.

Calling Program

```
10 COMMON (RESERV) STRING A = 10, STRING B, BYTE C
```

Subprogram

```
10 COMMON (RESERV) STRING A1 = 4, STRING A2 = 6, STRING B, BYTE C
```

In the calling program, A is a 10-character string. In the subprogram, A is subdivided into A1, which contains the first four characters, and A2, which contains the next six characters.

Each numeric variable in a common block or map should start on a word boundary. If the total storage allocation preceding the numeric variable is an odd number of bytes, use the FILL keyword to align the numeric variable on a word boundary. For example:

```
COMMON (RESERV) STRING A = 9, STRING FILL = 1, STRING B, REAL C
```

String variables and byte numerics in a common block or map can start on any byte boundary; however, numeric variables other than byte integers must start on a word boundary. For example:

```
10 MAP (RESERV) STRING A = 3, WORD X
```

If a program contains this line, BASIC–PLUS–2 signals the error "%Unaligned COM or MAP variable X in (RESERV)." This warning error tells you that BASIC–PLUS–2 added one byte of fill; BASIC–PLUS–2 moves the variable to the next word boundary in the resulting object file.

In a single program module, the size of a common block PSECT containing multiple common blocks of the same name is the sum of the lengths of each common area. The size of a map PSECT containing multiple maps of the same name is the length of the longest single map area. The order of variables in the common block and the order of multiple common blocks of the same name, determine the order of values in the shared area. For example:

```
10 COMMON (COM01) STRING A = 10
20 COMMON (COM01) WORD AA, BB, CC, DD, EE
30 MAP (MAP01) STRING B = 10
40 MAP (MAP01) WORD LL, MM, NN, OO, PP
```

These COMMON statements reserve 20 bytes of storage: 10 bytes for string A and two bytes for each of five word integers. The MAP statements reserve a total of 10 bytes; the first map reserves 10 bytes for string B, and the second map uses the same 10 bytes for five word integers.

When sharing map and common areas among program modules, you must make sure that the map or common PSECT is not overwritten when a subprogram is loaded into memory during program execution. You can ensure this by placing the map or common block in the root module. See Section 11.7 for more information on overlaying common and map areas.

11.6.2 Files

You can open and access a file in a calling program or subprogram. If you open the file in the calling program, you do not need to reopen the file in the subprogram to access the data. Files remain open until you open another file on that channel, close the file, or until the main program END statement is executed.

Because there is a single set of record pointers for all program modules, accessing a file on a given channel retrieves the next record, whether you perform the access from the main program or a subprogram. Similarly, the RESTORE # statement resets the record pointer to the first record in the file whether RESTORE # is executed in the main or subprogram.

This example accesses a RELATIVE file in a main and subprogram.

Main Program: SCAN

```
5       ON ERROR GOTO 19000
        MAP (EMPDAT) INTEGER EMP.NUM,          &
                     STRING NA.ME = 30%,       &
                     WAGE.CLASS = 2%,          &
                     JOB.TITLE = 20%,          &
                     REVIEW.DATE = 8%
        OPEN "EMP.DAT" FOR INPUT AS FILE #6%,   &
                ORGANIZATION RELATIVE VARIABLE, &
                ACCESS MODIFY,                  &
                MAP EMPDAT
           REC.NUM% = 1%
   START:  GET #6%, RECORD REC.NUM%
   CHECK:  IF WAGE.CLASS = "01"
             THEN
                CALL REVIEW(REC.NUM%)
                GOTO CHECK
             ELSE
                REC.NUM% = REC.NUM% + 1%
                GOTO START
           END IF
```

```
19000       IF ERR = 11%
              THEN
                 CLOSE #6%
                 PRINT 'Finished'
                 RESUME 32767
              ELSE
                 ON ERROR GOTO 0
            END IF
32767    END
```

Subprogram: REVIEW

```
10      SUB REVIEW ( INTEGER N% )
        ON ERROR GO BACK
        MAP (EMPDAT) STRING Z = 64%
        OPEN "REVREP.DAT" AS FILE #1%, ACCESS APPEND
        GET #6%, RECORD N%
        PRINT #1%, Z
        CLOSE #1%
32767   END SUB
```

The calling program opens an existing relative file, using the EMPDAT map. As the program retrieves records, it checks the WAGE_CLASS field for a value of "01". If it finds that value, it calls the subprogram REVIEW, passing the record number as a parameter.

The REVIEW subprogram does the following:

1. Receives the record number as a parameter
2. Executes an ON ERROR GO BACK, specifying that the calling program handles any errors
3. Remaps EMPDAT as a 64-character string variable
4. Opens the terminal-format file REVREP.DAT with ACCESS APPEND

REVIEW retrieves the record specified by the parameter *N%* and prints the variable *Z* to the terminal format file. Control then returns to the calling program so it can scan for another record.

11.7 Building Task Images

In order for a multi-module program to execute, the Task Builder must link the object files from each module into a single executable image. The Task Builder requires both a command file and an overlay descriptor language (ODL) file to create an executable image from multiple object files. The command file specifies input and output files. The ODL file specifies the way in which the modules in your program overlay one another.

In the BASIC–PLUS–2 environment, the BUILD command creates both a Task Builder command file and an ODL file. For example:

```
BASIC2
OLD MAIN
BASIC2
COMPILE
BASIC2
OLD SUB1
BASIC2
COMPILE
BASIC2
BUILD MAIN,SUB1
BASIC2
EXIT
$ TKB @MAIN
$ RUN MAIN
```

This command sequence does the following:

1. The first OLD command reads MAIN.B2S into memory.
2. The first COMPILE command compiles MAIN, creating an object module MAIN.OBJ.
3. The second OLD command reads SUB1.B2S into memory.
4. The second COMPILE command compiles SUB1, creating an object module SUB1.OBJ.
5. The BUILD command creates two files: MAIN.CMD and MAIN.ODL.
6. The EXIT command causes BASIC–PLUS–2 to exit, returning to monitor command level.
7. The TKB command invokes the Task Builder by using MAIN.CMD and MAIN.ODL as input files for the link. The Task Builder creates one executable image from the object modules.
8. The RUN MAIN command executes the image.

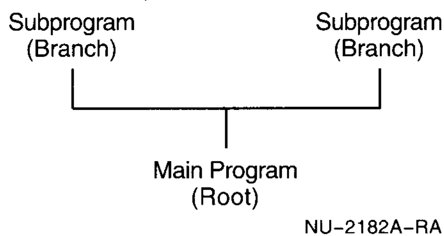
The Task Builder does the following:

1. Combines the object modules generated by the COMPILE command into a single, executable task image
2. Searches the library to resolve global references made by the program

The ODL file created by the BUILD command tells the Task Builder to concatenate all modules—that is, to perform no overlays. If the resulting executable image is too large to execute, you must edit the ODL file to specify an overlay structure.

The overlay structure is the way program code is brought into memory as the program executes. Changing the overlay structure of a task lets you decrease the amount of memory space for your task. Figure 11–1 illustrates the overlay structure.

Figure 11–1 Tree Figure Representing the Overlay Structure



The ODL file defines the root and branches in the task image. The root is the portion of the task that remains in memory throughout task execution. It includes the code for the main program, data local to the main program, data shared between program modules, and the object library modules needed to resolve the symbols in the generated code. The branches are the region of memory that contains the program code for subprograms, data local to the subprogram, and the object library modules needed to resolve symbols not already resolved in the main program.

In the following example, the main program calls two subprograms, SUB1 and SUB2, and SUB2 calls SUB3.

Main Program

```
10      PRINT 'This program adds and subtracts'
        PRINT 'two numbers'
        DECLARE REAL A, B
20      CALL SUB1 ( A, B )
        CALL SUB2 ( A, B )
32767   END
```

Subprogram 1

```
10      SUB SUB1 ( REAL X, Y )
        INPUT 'Input two numbers'; X, Y
32767   END SUB
```

Subprogram 2

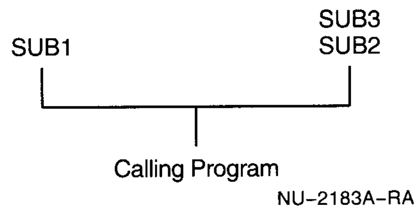
```
10      SUB SUB2 ( REAL X, Y )
        X_PLUS_Y = X + Y
        X_MINUS_Y = X - Y
20      CALL SUB3 ( X_PLUS_Y, X_MINUS_Y )
32767   END SUB
```

Subprogram 3

```
10      SUB SUB3 ( REAL W, V )
        PRINT 'A + B =' ; W
        PRINT 'A - B =' ; V
32767   END SUB
```

Figure 11–2 shows one possible relationship between the calling program or root segment, and the subprograms or branch segments.

Figure 11–2 Example of Overlay Structure



Each branch of the tree represents a program segment. Parallel branches at the same level represent program segments whose instructions and data are overlaid in memory as the program executes.

Figure 11–3 compares the amounts of memory space the task needs if the four programs are included in the root of the ODL file and if the subprograms are included in the branches of the ODL file.

Figure 11–3 Nonoverlay and Overlay Memory Requirements

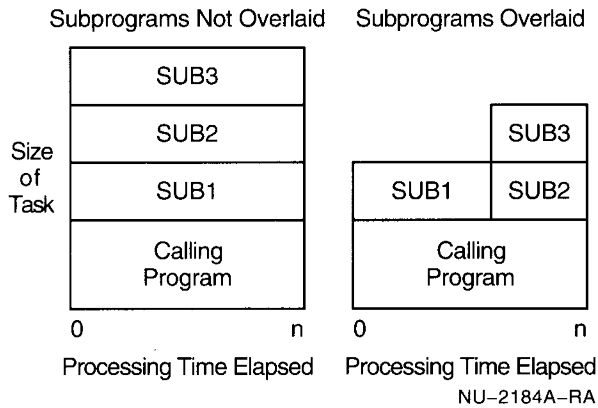


Figure 11–3 shows how large the task is at a given time during processing and what parts of the program are in memory at that time. Comparing them, you can see the nonoverlaid version needs more memory than the overlaid version. In the figure that shows subprograms overlaid, SUB2 and SUB3 overlay the memory reserved for SUB1 as the program executes.

To change the overlay structure defined for a task, you must change the contents of the ODL file before you link your program. To do this, edit the ODL file. See the *RSTS/E Task Builder Reference Manual* or the *RSX–11M/M–PLUS Task Builder Reference Manual* for more information about ODL files.

The ODL file for the task illustrated above where the subprograms are not overlaid looks like this:

```
        .ROOT USER
USER:   .FCTR SY:MAIN-SUB1-SUB2-SUB3-LIBR
LIBR:   .FCTR LB:BP2OTS/LB
        .END
```

You can edit your ODL file to overlay your subprograms, reducing the amount of memory space required for the task:

```
        .ROOT USER
USER:   .FCTR SY:MAIN-LIBR-*(BR1,BR2)
BR1:    .FCTR SY:SUB1-LIBR
BR2:    .FCTR SY:SUB2-SUB3-LIBR
LIBR:   .FCTR LB:BP2OTS/LB
        .END
```

The ODL file defines the root portion of the task:

```
USER:   .FCTR SY:MAIN-LIBR-*(BR1,BR2)
```

and two overlaid branches:

```
.FCTR SY:SUB1-LIBR
.FCTR SY:SUB2-SUB3-LIBR
```

The main program is concatenated with the BASIC–PLUS–2 disk library and the two subprograms, BR1 and BR2. Check your file to make sure the autoload operator (*) is included in the file. The autoload operator tells the Task Builder to generate autoload code to automatically load the appropriate program segment into memory as the program executes. See the *RSTS/E Task Builder Reference Manual* or the *RSX–11M/M–PLUS Task Builder Reference Manual* for more information on the autoload operator.

Check your ODL file to make sure that each branch and root segment is concatenated with the BASIC–PLUS–2 disk library so that the Task Builder can resolve global symbols such as thread names. Failure to concatenate object modules with the disk library can cause the Task Builder to signal the error "?Undefined symbol."

If you intend to use maps or common blocks to share data between main programs and subprograms, make sure the common block or map is defined in the main program. The main program is located in the root of the ODL file, and the data contained within is not overlaid.

For more information on linking and overlay procedures, see the *RSTS/E Task Builder Reference Manual* or the *RSX–11M/M–PLUS Task Builder Reference Manual*.

You can also run multi-module programs in the BASIC–PLUS–2 environment. To do this you must compile each subprogram, load the resulting object modules into memory with the LOAD command, read the main program in memory, and RUN the main program. For example:

```
BASIC2
OLD SUB1
BASIC2
COMPILE
BASIC2
OLD SUB2
BASIC2
COMPILE
BASIC2
LOAD SUB1+SUB2
BASIC2
OLD MAIN
BASIC2
RUN
```

The COMPILE command generates an object module for each program module. The main program is compiled when the RUN command is executed. Note that the loaded object modules are concatenated and not overlaid.

11.8 Non-BASIC–PLUS–2 Subprograms

Accessing subprograms written in languages other than BASIC–PLUS–2 can be useful. For example, MACRO subprograms have these advantages over BASIC–PLUS–2 subprograms:

- You can pass more parameters to a MACRO subprogram than to a BASIC–PLUS–2 subprogram.
- Some MACRO subprograms run faster than comparable BASIC–PLUS–2 subprograms.
- MACRO subprograms let you perform tasks that are difficult or impossible with BASIC–PLUS–2.

However, BASIC–PLUS–2 does not support the following operations in MACRO subprograms:

- Performing I/O or monitor operations
- Accessing virtual arrays or other kinds of files
- Creating strings or altering the lengths of existing strings

These operations may overwrite portions of the main program with subprogram instructions and data. They can cause the main program to abort or generate unpredictable results. Therefore, you should use extreme care when coding non-BASIC–PLUS–2 subprograms.

The following sections use MACRO as an example of a non-BASIC–PLUS–2 language because of its widespread use and versatility.

11.8.1 Parameter-Passing Mechanisms

Different languages can use different methods for receiving and storing parameters. Therefore, calling non-BASIC–PLUS–2 subprograms from a BASIC–PLUS–2 program module may require special *parameter-passing mechanisms*.

The parameter-passing mechanism refers to the way in which data is passed to a subprogram. You can pass parameters in the following ways:

- By immediate value
- By reference
- By descriptor

BASIC–PLUS–2 supports these methods with the three BY clauses to the CALL statement:

- BY VALUE specifies that BASIC–PLUS–2 passes a value to the subprogram.
- BY REF specifies that BASIC–PLUS–2 passes the address of a value to the subprogram.
- BY DESC specifies that BASIC–PLUS–2 passes the address of a string or array descriptor to the subprogram. These descriptors include pointers to the data.

The parameter-passing mechanism determines whether a subprogram can modify the parameter. If you send a parameter BY VALUE, the subprogram does not receive the address of the parameter and therefore cannot modify it. On the other hand, parameters passed BY REF or BY DESC can be modified because the subprogram receives either the address of the parameter or the address of the parameter's descriptor.

Table 11–1 shows allowable parameters for each parameter-passing mechanism.

Table 11–1 BASIC–PLUS–2 Parameter-Passing Mechanisms

Parameter	BY VALUE	BY REF	BY DESC
	Integer and Real Data		
Variables	Yes[2]	Yes[1]	No
Constants	Yes[2]	Local copy[1]	No
Expressions	Yes[2]	Local copy[1]	No
Elements of a nonvirtual array	Yes[2]	Local copy[1]	No
Virtual array elements	Yes[2]	Local copy[1]	No
Nonvirtual entire array	No	Yes	Yes[1]
Virtual entire array	No	No	Yes
	String Data		
Variables	No	Yes	Yes[1]
Constants	No	Local copy	Local copy[1]
Expressions	No	Local copy	Local copy[1]
Nonvirtual array elements	No	Local copy	Local copy
Virtual array elements	No	Local copy	Local copy[1]
Nonvirtual entire arrays	No	Yes	Yes[1]
Virtual entire arrays	No	No	Yes[1]

[1]Specifies the default parameter-passing mechanism

[2]Two asterisks indicate that the value can have 16 bits, at most

The default parameter-passing mechanisms for the CALL statement correspond precisely to the way a BASIC–PLUS–2 subprogram expects to receive the parameters.

If a BY clause appears before the parameter list, that parameter-passing mechanism applies to all parameters in the list; however, this can be overridden by another BY clause after an individual parameter.

If the parameter list contains two commas with no expression between them, BASIC–PLUS–2 passes a null argument in that position. For example:

```
2000 CALL MACRO1 BY REF (SS_STRING$ BY DESC, , , 35% BY VALUE)
```

This statement calls a MACRO subprogram specifying the following:

- The first parameter is a string, passed by descriptor
- The second and third parameters are null
- The fourth parameter is an integer constant, passed by value

Specifying a null parameter is the same as specifying –1% BY VALUE for that parameter, and BASIC–PLUS–2 places this value in the argument list. You cannot pass null parameters to a BASIC–PLUS–2 subprogram.

The BY VALUE parameter-passing mechanism is recommended only for system directives and subprograms written in MACRO–11. Moreover, you should use BY VALUE only for values that can fit in a 16-bit word. This is because the argument list allows only 16 bits per argument.

11.8.2 Declaring Non-BASIC–PLUS–2 Subprograms

When calling non-BASIC–PLUS–2 subprograms, you should always declare them with the EXTERNAL statement. The format of the EXTERNAL statement is as follows:

```
                              { BY VALUE }
EXTERNAL SUB { sub-name [     { BY REF   } ]
                              { BY DESC  }

                 { BY VALUE }
[ [data-type] [  { BY REF   } ], . . . ]}, . . .
                 { BY DESC  }
```

sub-name
Is the 1- to 6-character name of the subprogram. A subprogram cannot have the same name as any other subprogram, MAP, or COMMON within the same task. The name can be a quoted or unquoted string but cannot be a string variable.

BY VALUE
BY REF
BY DESC
Specifies the parameter-passing mechanism. A BY clause preceding the parameter list specifies the default passing mechanism. A BY clause appearing in the parameter list applies only to the particular parameter it follows.

data-type
Is a data-type keyword. See Chapter 7 for more information about data-type keywords.

By declaring the non-BASIC–PLUS–2 subprogram, you need to specify the data types and parameter-passing mechanisms only once. After the subprogram is declared, BASIC–PLUS–2 automatically converts parameters to the proper data type and specifies the proper parameter-passing mechanism each time the subprogram is called. For example:

```
100   DECLARE WORD XYZ, STRING ABC
200   EXTERNAL SUB MACR01(LONG BY VALUE, STRING BY DESC)
   .
   .
   .
5000  CALL MACR01 BY REF (XYZ, ABC)
```

Line 100 declares two program variables: a WORD integer XYZ and a dynamic string ABC. The EXTERNAL statement at line 200 declares MACR01 as a subprogram with two parameters: a LONG integer and a string. When MACR01 is called at line 5000, BASIC–PLUS–2 automatically passes the value contained in XYZ as a LONG integer because this is the data type specified in the EXTERNAL statement.

Note that in a CALL using parameters passed by reference, if the actual parameter's data type does not match that specified in the EXTERNAL statement, BASIC–PLUS–2 signals the error "?Mode for parameter of routine changed to match declaration." This tells you that BASIC–PLUS–2 has made a local copy of the value of the parameter and that this local copy has the data type specified in the EXTERNAL declaration. BASIC–PLUS–2 warns you of this only when the change means that the parameter can no longer be modified by the subprogram.

11.8.3 Calling Non-BASIC–PLUS–2 Subprograms

To transfer control from a BASIC–PLUS–2 main program to a non-BASIC–PLUS–2 subprogram, use the CALL statement.

The CALL statement passes parameters to the subprogram either by reference, by value, or by descriptor, depending on the type of parameter and the BY clause. The format of the CALL statement is as follows:

```
                  { BY VALUE }              { BY VALUE }
CALL sub-name [   { BY REF   } ] [ param [  { BY REF   } ] , . . . ]
                  { BY DESC  }              { BY DESC  }
```

sub-name
Is the 1- to 6-character name of the non-BASIC–PLUS–2 subprogram. A subprogram cannot have the same name as any other subprogram, MAP, or COMMON within the same task. The name can be a quoted or unquoted string but cannot be a string variable.

BY VALUE
BY REF
BY DESC
Specifies the parameter-passing mechanism. A BY clause preceding the parameter list specifies the default passing mechanism. A BY clause appearing in the parameter list applies only to the particular parameter it follows.

param
Is an argument, or parameter, passed from the main program to the subprogram. The number of parameters you can include is limited by the size of the temporary storage area (stack) allocated in your task image. The maximum number of parameters you can pass to a MACRO subprogram is 255. For example:

```
300  CALL SUBPRG BY REF (A$,B%,C$(1%,1%))
```

Note

The Task Builder requires a name containing only members of the RAD-50 character set. See C for information on the RAD-50 character set. The subprogram name can begin with a dollar sign ($) only if the name is enclosed in quotes. It is recommended that you do not begin subprogram names with a dollar sign, as names of this form are reserved for DIGITAL.

11.9 MACRO Subprograms

When calling MACRO subprograms, the name in the CALL statement must correspond to the name of the MACRO subprogram, as defined by the subprogram's global entry-point label. This can be different from the name of the file containing the code and the name defined by the .TITLE assembly directive; however, if the subprogram includes only a single entry point, you should use the same name for the entry-point label and the title. For example, suppose a main program includes the following statement:

```
50 CALL INSRT
```

The code of the MACRO subprogram should include the title and label-name INSRT, as in the example below:

```
        .TITLE  INSRT

; MODULE FUNCTION:
;       THIS MODULE DEMONSTRATES
;       THE FORMAT FOR MACRO SUBPROGRAMS

        .PSECT INSRT
INSRT::                   ; SUBPROGRAM NAME
        RTS  PC           ; RETURN TO MAIN PROGRAM
        .END
```

The RTS PC (or RETURN) instruction returns control to the calling program. It corresponds to the SUBEND statement of a BASIC–PLUS–2 subprogram.

For more information on linking and overlay procedures, see the *RSTS/E Task Builder Reference Manual* or the *RSX–11M/M–PLUS Task Builder Reference Manual.*

11.9.1 Passing Parameters

You can pass data from a BASIC–PLUS–2 main program to a MACRO subprogram by including parameters in the CALL statement of the main program. BASIC–PLUS–2 imposes two restrictions on the kinds of parameters that main programs can pass to MACRO subprograms:

- BASIC–PLUS–2 main programs cannot pass virtual arrays to MACRO subprograms.
- MACRO subprograms cannot change the length of strings passed to them as parameters by BASIC–PLUS–2 main programs, nor can they create new strings.

If you need to access virtual arrays or change string lengths in a MACRO subprogram, place the string and array data in a COMMON or MAP area before calling the subprogram. Once the subprogram has performed its operations on the data and has returned control to the main program, the main program can move the data back into arrays and dynamic strings. For information on using COMMONs and MAPs with MACRO subprograms, see Section 11.9.2.

MACRO subprograms do not have SUB statements to define the parameters they receive. Therefore, in order to access those parameters, you need to know where the main program stores them.

Like BASIC–PLUS–2 subprograms, MACRO subprograms that receive parameters from a main program receive an argument list containing information about those parameters. The information in the argument list varies, depending on whether a given parameter is passed by reference or by descriptor:

- The first word of the argument list always contains, in its low-order byte, the number of arguments in the list. The high-order byte in this word is undefined.
- Each word after the first contains a pointer to one of the parameters in the CALL statement, in the same order as the parameters appear in the CALL statement. If you pass a parameter by reference, the subprogram receives in the argument list the address where that parameter, or a local copy of it, is located. If you pass a parameter by descriptor, the subprogram receives in the argument list the address of a descriptor block. The descriptor block contains information about the parameter or its local copy, including the address where the parameter or copy is located and the length of the parameter. See C for more information about descriptor blocks.

The argument lists look like the illustration in Figure 11–4.

Whenever BASIC–PLUS–2 encounters a CALL statement, it stores the address of the argument list's first word in general register R5. Therefore, subprograms can express parameter addresses as offsets from the value stored in register R5.

BASIC–PLUS–2 passes most parameters by reference; however, it automatically passes certain parameters, such as string data and arrays, by descriptor. When you specify the BY REF clause, BASIC–PLUS–2 passes all parameters to the MACRO–11 subprogram by reference. Therefore, all the addresses in the argument list generated by a CALL statement with a BY REF clause refer to

Figure 11–4 Argument List Format

Undefined	Number of arguments
Parameter #1	
Parameter #2	
. . .	
Parameter #n	

NU-2185A-RA

the parameters themselves or to local copies of them. If the subprogram needs to know a string length, you must pass that information as a parameter. For example, a main program might contain the following CALL statement:

```
19000 CALL MACSUB BY REF (STR.NG$,LEN.STR%,A$(1%,5%))
```

This statement would generate an argument list like the following:

```
ADDRESS   VALUE

022060    000003
022062    004560     ARGUMENT LIST
022064    005632
022066    035766
```

The actual addresses and the values stored in them depend on your program. In this example, memory location 22060 contains the octal value 000003, since there are three parameters in the argument list. Locations 22062 and 22064 contain the addresses of string variable STR.NG$ and integer variable LEN.STR%. Location 22066 contains the address of a local copy of string array element A$(1%,5%).

General register R5 contains the value 22060, the address of the first word in the argument list. Therefore, to access the parameters, define their locations as offsets from R5. For example, use this MACRO statement to move the first two bytes of variable STR.NG$ into general register R1:

```
MOV @2(R5),R1            ; SET R1 = STR.NG$
```

The following example shows how a subprogram accesses and modifies parameters passed by reference from a BASIC–PLUS–2 main program. Note that the parameters passed in the CALL BY REF statement include the lengths of strings A$ and B$.

BASIC–PLUS–2 Main Program

```
10      PRINT 'This program writes substring B$ into string A$,'
        PRINT 'starting at character position C.'
        PRINT
   ASK: INPUT 'Enter string A$';A$
        GOTO 32767 IF A$ = 'DONE'
        INPUT 'Enter sub-string B$';B$
        INPUT 'Enter C ';C%
        CALL INSRT BY REF (A$,LEN(A$),B$,LEN(B$),C%)
        IF C% = 0%
            THEN PRINT 'New value of A$ is ';A$
            ELSE PRINT 'Attempt unsuccessful'
        END IF
        PRINT
        PRINT 'Do you want to continue?'
        PRINT 'If not, type "(DONE)".'
        PRINT
           GOTO ASK
32767   END
```

MACRO Subprogram

```
      .TITLE  INSRT
      .IDENT /01/
; MODULE FUNCTION:
;      THIS SUBPROGRAM WRITES SUBSTRING B$
;      INTO STRING A$, BEGINNNING AT CHARACTER C%

;
;LOCAL MACROS:
;
;
; LOCAL DATA BLOCKS:
;
      .PSECT  DATA,D,RW
;
; LOCAL OFFSETS:

A    =  2.
LNA  =  4.
B    =  6.
LNB  =  8.
C    = 10.
```

```
;
; FUNCTION DETAILS:
;
;      INPUTS:
;            ARG1 = ADDRESS OF A$
;            ARG2 = ADDRESS OF LENGTH OF A$
;            ARG3 = ADDRESS OF B$
;            ARG4 = ADDRESS OF LENGTH OF B$
;            ARG5 = ADDRESS OF C%

;      OUTPUTS:
;            C% = 0 IF OPERATION SUCCESSFUL
;            C% = -1 IF OPERATION UNSUCCESSFUL
       .PAGE
       .SBTTL
       .PSECT INSRT
INSRT::
       MOV     @C(R5),R2        ; R2 = C%
       BLE     ERREX            ; GOTO ERREX IF C <= 0
       ADD     @LNB(R5),R2      ; R2 = C% + LEN(B$)
       DEC     R2               ; MAKE R2 A LENGTH
       CMP     R2,@LNA(R5)      ; DOES B$ FIT INTO A$?
       BGT     ERREX            ; IF NOT, GOTO ERREX
       MOV     A(R5),R0         ; R0 = ADDRESS OF A$
       MOV     @C(R5),R2        ; SET R2 = C%
       DEC     R2               ; SET R2 = C% - 1
       ADD     R2,R0            ; SET R0 = ADDRESS OF A$ + C%
       MOV     @LNB(R5),R2      ; SET R2 = LEN(B$)
       BEQ     ERREX            ; GO TO ERREX IF LEN(B$) = 0
       MOV     B(R5),R1         ; SET R1 = ADDRESS OF B$
1$:
       MOVB     (R1)+,(R0)+     ; INSERT CHAR FROM B$ INTO A$
       SOB     R2,1$            ; REPEAT FOR REMAINING CHARACTER
       CLR     @C(R5)           ; SUCCESS. SET C% = 0
       RTS     PC               ; RETURN TO MAIN PROGRAM
ERREX:
       MOV     #-1,@C(R5)       ; FAILURE. SET C% = -1
       RTS     PC               ; RETURN TO MAIN PROGRAM
       .END
```

Compile the main program with the BASIC–PLUS–2 compiler. Assemble the subprogram with the MACRO assembler. Then build and run them as you would any multi-segment BASIC–PLUS–2 task, by including the MACRO object module name with the main program name in the BUILD command.

When you run the program, it returns the following:

```
$ RUN MNPROG

This program writes substring B$ into string A$,
starting at character position C.

Enter string A$? ABCDEF
Enter sub-string B$? XYZ
Enter C? 1

New value of A$ is XYZDEF

Do you want to continue? DONE
```

Note

Unmatched parameter boundaries or data types in the main program and subprogram can cause the error "?Odd address trap" or "?Memory management violation."

Simple variables and entire arrays are modifiable parameters. For example, the previous subprogram changes A$ and then returns the changed value to the main program.

All constants, expressions, and array elements, however, are nonmodifiable. That is, when a subprogram receives these parameters, the addresses in the argument list point to local copies of the parameters rather than to the actual data. A MACRO subprogram can change local-copy values, but such changes do not affect the constants, expressions, and arrays of the main program.

You can pass local copies of simple variables and entire arrays by enclosing the individual variable and array names in parentheses. In the following example, A$ is a modifiable parameter but B$ is nonmodifiable.

```
19000 CALL MACSUB BY REF (A$,(B$))
```

When the CALL parameter is a string, the corresponding value in the argument list points to a 2-word descriptor block. The first word of this block contains the address of the first byte in the string. The second word expresses the length of the string, in bytes.

For example, suppose a main program includes the statement:

```
50 CALL INSRT BY REF (A$ BY DESC, B$ BY DESC, C% BY REF)
```

This generates a 4-word argument list:

```
ADDRESS   VALUE

022060    000003
022062    004532     ARGUMENT LIST
022064    023462
022066    026534
```

The actual addresses, and the values stored in them, depend on your program. In this example, the second word in the argument list contains the address of the descriptor block for A$. If A$ were a 6-byte string, such as ABCDEF, the descriptor block would look like this:

```
ADDRESS   VALUE

004532   020652     DESCRIPTOR BLOCK
004534   000006
```

If you include an array in a CALL statement, the argument list contains the address of the second word of the descriptor block. This word in turn contains the address of the first element in the array. (The first word in the descriptor block is the array descriptor word, which defines the array type and length. See C for more information on the array descriptor word.)

This example shows how a MACRO subprogram can access parameters passed by descriptor.

BASIC–PLUS–2 Main Program

```
10      PRINT 'This program writes substring B$ into string A$"
        PRINT 'starting at character position C.'
        PRINT
   ASK: INPUT 'Enter string A$';A$
        IF A$ = "DONE" GOTO 32767
        INPUT 'Enter sub-string B$';B$
        INPUT 'Enter C ';C%
        CALL INSRT BY REF (A$ BY DESC, B$ BY DESC, C% BY REF)
        IF C% = 0%
           THEN PRINT 'New value of A$ is ';A$
           ELSE PRINT 'Attempt unsuccessful'
        END IF
        PRINT
        PRINT 'Do you want to continue?'
        PRINT 'If not, type "DONE".'
           GOTO ASK
32767   END
```

MACRO Subprogram

```
      .TITLE INSRT
      .IDENT /01/

;MODULE FUNCTION:
;     THIS SUBPROGRAM OVERWRITES SUBSTRING B$
;     INTO STRING A$, BEGINNING AT CHARACTER C%.

;
; LOCAL MACROS:
;
;
; LOCAL DATA BLOCKS:
;
      .PSECT  DATA,D,RW

;
; LOCAL OFFSETS:
A = 2.
B = 4.
C = 6.
;
; FUNCTION DETAILS:
;
;     INPUTS:
;          ARG1 = ADDRESS OF A$ STRING DESCRIPTOR
;          ARG2 = ADDRESS OF B$ STRING DESCRIPTOR
;          ARG3 = ADDRESS OF C%
;     OUTPUTS:
;          C% = 0 IF SUCCESSFUL
;          C% = -1 IF UNSUCCESSFUL

      .PAGE
      .SBTTL
      .PSECT INSRT
INSRT::
      MOV    A(R5),R0           ; SET R0 = ADDRESS OF A$ DESC
      MOV    B(R5),R1           ; SET R1 = ADDRESS OF B$ DESC
      MOV    @C(R5),R2          ; SET R2 = C%
      BLE    ERREX              ; GO TO ERREX IF C <= 0
      ADD    2(R1),R2           ; SET R2 = C% + LENGTH OF B$
      DEC    R2                 ; MAKE R2 A LENGTH
      CMP    R2,2(R0)           ; DOES B$ FIT INTO A$?
      BGT    ERREX              ; IF NOT, GO TO ERREX
      MOV    (R0),R0            ; SET R0 = ADDRESS OF A$
      MOV    @C(R5),R2          ; SET R2 = C%
      DEC    R2                 ; SET R2 = C% -1
      ADD    R2,R0              ; SET R0 = ADDRESS OF FIRST CHAR
      MOV    2(R1),R2           ; SET R2 = LENGTH OF B$
      BEQ    ERREX              ; IF B$ = 0, GO TO ERREX
      MOV    (R1),R1            ; SET R1 = ADDRESS OF B$
```

```
1$:
      MOVB    (R1)+,(R0)+          ; INSERT A CHAR FROM B$ INTO A$
      SOB     R2,1$                ; REPEAT
      CLR     @C(R5)               ; SUCCESS. SET C% = 0
      RTS     PC                   ; RETURN TO MAIN PROGRAM
ERREX:
      MOV     #-1,@C(R5)           ; FAILURE. SET C% = -1
      RTS     PC                   ; RETURN TO MAIN PROGRAM
      .END
```

In this example, the lengths of A$ and B$ are part of the descriptor blocks instead of being passed as parameters in the CALL statement.

11.9.2 Common Blocks and Maps

BASIC–PLUS–2 allows you to access COMMON and MAP areas from MACRO subprograms. This enables you to share large amounts of data between your BASIC–PLUS–2 main programs and your MACRO subprograms. In addition, you can use MACRO subprograms to initialize COMMON or MAP areas in BASIC–PLUS–2 main programs.

You can rewrite the main program and subprogram in the previous example so that they share data by means of a MAP statement rather than by passing parameters in the CALL statement.

Main Program

```
10      MAP (RESERV) STRING A, INTEGER LNA,     &
                     STRING B=8%, INTEGER LNB,  &
                     INTEGER C
   ASK: PRINT 'This program writes substring B into string A'
        PRINT 'starting at character position C.'
        PRINT
        INPUT 'Enter 16-character string A';A
        INPUT 'Enter 8-character sub-string B';B
        INPUT 'Enter C ';C
        LNA = LEN(A)
        LNB = LEN(B)
        CALL INSRT
        IF C = 0
           THEN PRINT 'New value of A is ';A
           ELSE PRINT 'Attempt unsuccessful'
        END IF
        PRINT
        PRINT 'Do you want to continue?'
        PRINT 'If not, type "DONE".'
           GOTO ASK
32767   END
```

The MAP statement in this program sets the size of the A and B strings to a predetermined 16 and 8 bytes.

When the BASIC–PLUS–2 compiler generates object code from the BASIC–PLUS–2 source program, it creates PSECTs for all COMMON and MAP statements with the same name. You can see this most clearly if you compile the previous BASIC–PLUS–2 source program with the COMPILE /MACRO command. The compiler generates the following MACRO code for the MAP statement in that program:

```
      .PSECT RESERV,RW,D,GBL,REL,OVR
RESERV:
      .PSECT RESERV
      .BLKW 15.
```

In order for the MACRO subprogram to access the data stored in MAP area RESERV, you have to define a subprogram PSECT of the same name and attributes as that created by the MAP statement of the main program. The subprogram can then assign variable names to the data in that PSECT, and use those variables as operands in its instructions. For example:

```
      .TITLE INSRT
      .IDENT /01/

;
;MODULE FUNCTION:
;     THIS SUBPROGRAM USES A MAPPED AREA
;     TO OVERWRITE SUBSTRING B INTO STRING A,
;     BEGINNING AT CHARACTER C.
;
; LOCAL MACROS:
;
;
; LOCAL DATA BLOCKS:
      .PSECT RESERV RW,D,GBL,REL,OVR
A:    .BLKB 16.
LNA:  .BLKW
B:    .BLKB 8.
LNB:  .BLKW
C:    .BLKW
;
; FUNCTION DETAILS:
;
;     INPUTS:
;          PSECT RESERV CONTAINS A,LNA
;          B,LNB, AND C
;
;     OUTPUTS:
;          C = 0 IF OPERATION WAS SUCCESSFUL
;          C = -1 IF OPERATION FAILED
;
```

```
; CODE BEGINS HERE:
      .PAGE
      .SBTTL
      .PSECT INSRT
INSRT::
      MOV     C,R2                ; SET R2 = C
      BLE     ERREX               ; IF C <= 0, GO TO ERREX
      ADD     LNB,R2              ; SET R2 = C + LEN(B)
      DEC     R2                  ; MAKE R2 A LENGTH
      CMP     R2,LNA              ; DOES B FIT INTO A?
      BGT     ERREX               ; IF NOT, GO TO ERREX
      MOV     C,R2                ; SET R2 = C
      DEC     R2                  ; SET R2 = C - 1
      MOV     #A,R0               ; SET R0 = ADDRESS OF A
      ADD     R2,R0               ; SET R0 = ADDRESS OF FIRST CHAR REPLACED
      MOV     LNB,R2              ; SET R2 = LENGTH OF B
      BEQ     ERREX               ; IF B = 0, GO TO ERREX
      MOV     #B,R1               ; SET R1 = ADDRESS OF B
1$:
      MOVB    (R1)+,(R0)+         ; INSERT CHARACTER FROM B IN A
      SOB     R2,1$               ; REPEAT
      CLR     C                   ; SUCCESS. SET C = 0
      RTS     PC                  ; RETURN TO MAIN PROGRAM
ERREX:
      MOV     #-1,C               ; FAILURE. SET C = -1
      RTS     PC                  ; RETURN TO MAIN PROGRAM
      .END
```

When you build this multi-segment task, the Task Builder defines a single area named RESERV. If the buffer allocations in the main program and subprogram differ, the Task Builder defines an area equal to the larger allocation.

The Task Builder does not check to see that the main program and subprogram define the same data types and boundaries in their common areas. If the areas do not correspond, BASIC–PLUS–2 signals the error “Odd address trap” or “Memory management violation” when you run the task. For this reason, be sure you properly align your data definitions in the main program and subprogram. Remember that COMMON statements of the same name within a single program module are concatenated. For example:

```
10 COMMON (RESERV) STRING VR.STR = 30%, STRING FX.STR = 30%
20 COMMON (RESERV) STRING A, INTEGER B
```

These statements generate a single PSECT:

```
       .PSECT RESERV,RW,D,GBL,REL,OVR
RESERV:
       .PSECT RESERV
       .BLKW 39.
```

The total area set aside for RESERV is 78 bytes, the sum of the two COMMON statements. Variable A begins at byte location 60 of RESERV, not at location 0. If these were MAP statements, the area set aside for RESERV would equal the larger allocation, that is, 30 words. Variables VR.STR and A, in that case, would both begin at byte location 0.

In using MAP or COMMON areas to share data between main programs and subprograms, remember the following:

- You must check the lengths of your string and integer elements to make sure that you correctly line up the areas reserved by your main and subprograms.
- You must reserve eight bytes of storage for each floating-point variable in the corresponding PSECT of your MACRO subprogram if you compile your BASIC–PLUS–2 program with double precision.
- You must assign the same name and attributes to a MAP or COMMON area of the main program and the corresponding PSECT of the subprogram.

11.9.3 Initializing COMMONs and MAPs

You can use MACRO routines to initialize COMMONs and MAPs in a BASIC–PLUS–2 main program. For example, a main program could begin with COMMON statements, in which it stored data that both the main program and subprogram want to use in printing error messages or checking maximum values. If you were to assign values to those COMMON areas by means of statements in the main program, the first lines of the BASIC–PLUS–2 source code would look like this:

```
10    COMMON (FIXSTR) STRING OUT.STR = 10%,    &
                      BAD.INFO = 24%,          &
                      ATLIN = 8%
20    COMMON (FIXDAT) WORD MAXNUM,             &
                      DOUBLE MAXVAL,           &
                      WORD BADNUM,             &
                      STRING FUN.STR = 6%
30    OUT.STR  = 'Output is'
      BAD.INFO = 'Bad information supplied'
      ATLIN    = ' At line'
40    MAXNUM   = 100
      MAXVAL   = 2E6
      BADNUM   = -1
      FUN.STR  = ' FUNNY'
```

In this example, seven statements are executed to initialize variables in the COMMON area. In addition, storage is allocated to each constant before they are placed in the COMMON, and none of this storage is recovered. The following MACRO code performs the same initialization procedure as the previous BASIC–PLUS–2 code:

```
     .TITLE INIT

; MODULE FUNCTION:
;    THIS MODULE INITIALIZES THE COMMON
;    AREAS OF THE MAIN PROGRAM

; INITIALIZE FIXSTR
     .ENABLE  LC                          ;ENABLE LOWER CASE
     .PSECT   FIXSTR,RW,D,GBL,REL,OVR
     .ASCII   /Output is /                ;OUT.STR$ len = 10
     .ASCII   /Bad information supplied/  ;BAD.INFO$ len = 24
     .ASCII   / at line/                  ;ATLIN$    len = 8
; INITIALIZE FIXDAT
     .PSECT   FIXDAT,RW,D,GBL,REL,OVR
     .WORD .  100.                        ;MAXNUM%
     .FLT4    2E6                         ;MAXVAL
     .WORD    -1                          ;BADNUM
     .ASCII   / FUNNY/                    ;FUN.STR$ len = 6
     .END
```

This routine is not, strictly speaking, a subprogram. The main program cannot call it, as it does not contain any executable statements. But, if you build this module into your task as though it were a subprogram, you can omit statements 30 and 40 in the main program. An initialization routine like this one, therefore, can save you both time and memory space when you run the task.

Note

Because this routine contains no code and is not a subprogram, you cannot call it later in the main program to reinitialize values in the COMMON.

11.9.4 Building Task Images

Follow this general procedure when you build your segmented task:

- Compile the BASIC–PLUS–2 modules and assemble the MACRO modules.

- Include in the BUILD command all the object modules you wish to combine into a single task image, or modify the BUILD-generated ODL file to include individual subprograms in the task. If your program includes RMS file operations, use the appropriate BUILD command qualifier to incorporate RMS–11 code into your task.
- Use the TKB command to link your program.

The ODL file generated by the BUILD command causes the Task Builder to concatenate all the modules in the root of the task. Since the operating system and hardware impose restrictions on task size, you may need to design an overlay structure for the task. For more information on overlay structures, see Section 11.7 and the *RSTS/E Task Builder Reference Manual* or the *RSX–11M/M–PLUS Task Builder Reference Manual.*

When designing an overlay structure for a task that includes either BASIC–PLUS–2 or MACRO subprograms, you should do the following:

- Think about overlay structure in the early stages of programming. Design your task to take advantage of overlays.
- Test each module separately, writing small programs to call the modules and supply whatever data they need.
- Be sure you know where global symbols will be resolved and when overlays will be brought into memory when you run the task.
- Be sure to align MAP and COMMON variables in the main program and subprograms.
- Use co-trees (that is, overlay structures with independent root segments) only when necessary.

Designing an overlay structure for a task that includes a MACRO subprogram is similar to designing an overlay structure for a task that includes a BASIC–PLUS–2 subprogram; however, a knowledge of MACRO may enable you to take advantage of BASIC–PLUS–2 threaded code to decrease task size and enhance performance.

When the BASIC–PLUS–2 compiler translates a source program into object code, it generates threaded, rather than in-line code. That is, the compiler generates from the BASIC–PLUS–2 source program a series of global symbols and arguments. These symbols are names of routines that perform the operations the user task requires. When you build the task image, the Task Builder resolves the global symbols by searching within the modules of the task itself, and within the BASIC–PLUS–2 object library and resident library, for the routines they refer to.

You can determine which global symbols the Task Builder will need to resolve if you compile your program with the /MACRO qualifier. For example:

```
20 PRINT A%
```

From this line, the compiler generates the following threaded code:

```
L20:    LIN$        ,20           ; #20
        CLI$S
        IPT$
        MOI$MS      ,$IDATA+800   ; A%
        PVI$SI      ,0            ; #0
        EOL$
```

The code generated when you compile a program with the /MACRO qualifier is not the same as MACRO code you might use when programming the task. Rather, it is the MACRO equivalent of the object code generated by the BASIC–PLUS–2 compiler. In the previous example, L20: is a label identifying this particular block of code, while such symbols as LIN$ and CLI$ are global symbols representing BASIC–PLUS–2 routines. The Task Builder resolves these global symbols by performing the following sequence of operations:

1. It searches within the module itself, and within other modules in the same segment, for a resolution of the symbol.
2. It searches in modules along the same branch toward the root, in the root module, and in the memory-resident library if there is one.
3. It searches in modules along the same branch away from the root.
4. It searches co-trees.
5. It searches in the BASIC–PLUS–2 object library.

Your design of the overlay structure for a task should reflect this resolution sequence. Otherwise, your task may not be executable. For example, suppose you design an overlay structure in which one subprogram calls a second subprogram. The second subprogram contains an undefined symbol that you expected the Task Builder to resolve by searching the BASIC–PLUS–2 object library; however, the Task Builder resolves that symbol by bringing a third subprogram into memory and overlaying that third subprogram before searching the BASIC–PLUS–2 object library. When the second subprogram is finished and attempts to return to the first subprogram, the task aborts with a "?Memory management violation" or "?Odd address trap" error.

You can avoid this problem, and at the same time conserve space in the task, by placing in the root any threads needed to resolve global symbols, especially potentially ambiguous ones. Inspect your task's map file to find the OTS module that contains the thread you need. Then edit the ODL file to put the module in the root segment. The ODL file in the following example, for

instance, shows how a RSTS/E system uses the $START module to force the string arithmetic into the root segment of the task MNPROG:

```
        .ROOT BASIC-PLUS-22-RMSROT-USER,RMSALL
USER:   .FCTR SY:MNPROG-LB:BP2OTS/LB:$START-LIBR-*(BR1,BR2)
BR1:    .FCTR SY:SUBPR1-LIBR
BR2:    .FCTR SY:SUBPR2-LIBR
LIBR:   .FCTR LB:BP2OTS/LB
@LB:BP2IC1
@LB:RMS11S
        .END
```

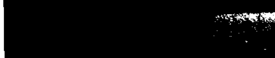
The file specifications in the ODL file vary, depending on the operating system. See the *RSTS/E Task Builder Reference Manual* or the *RSX-11M/M-PLUS Task Builder Reference Manual* for more information.

11.9.5 Handling Errors

MACRO subprograms should not contain error-handling routines that will abort the task. If a fatal error occurs in a MACRO subprogram, the subprogram should return control to the main program and signal to it that an error has occurred.

You can use parameters or common blocks to return status information to the calling program. Or, you can use ERR, ERL, and ERN$ functions in the program to determine the kind and source of error. The information returned by ERN$ and ERL differs, depending on whether the program uses a BY REF clause:

- If the program does not use a BY REF clause, ERN$ returns the name of the subprogram called, and ERL returns a value of 0.
- If the program uses a BY REF clause, ERN$ returns the name of the calling program, and ERL returns the line number of the CALL BY REF statement.
- If the program does not use a BY REF clause, you cannot use Ctrl/C trapping.



12
File Input-Output

This chapter explains the BASIC–PLUS–2 file organizations and record operations that are implemented through PDP–11 Record Management Services (RMS–11). For a more thorough understanding of file organization and file and record operations, see the RMS documentation for your system.

12.1 Introduction

When you open a file from a BASIC–PLUS–2 program and specify a sequential, relative, or indexed file organization with the OPEN statement, BASIC–PLUS–2 invokes Record Management Services (RMS–11) to handle the file. If the file already exists, RMS–11 checks to see that the file's attributes match the attributes you specify in the OPEN statement. For new files, RMS–11 defines a file structure and writes a label describing that structure.

The advantages of using RMS–11 are as follows:

- RMS–11 has the ability to perform more complex file operations, such as the use of indexed files
- RMS–11 protects you against overwriting old records with new records
- RMS–11 is standardized so you can easily transport your files to different operating systems

RMS stores data in physical *blocks*. A block is the smallest number of bytes BASIC–PLUS–2 transfers in a read or write operation. On disk, a block is 512 bytes. On magnetic tape, it is between 18 and 8192 bytes.

RMS stores one or more *data records* in each block. A data record can also be divided into smaller units, called *fields*. A data record can be smaller than, equal to, or larger than a disk block.

If you open a file without specifying a file name or an ORGANIZATION clause, the operating system describes the file structure and handles file operations. On RSTS/E systems, these files are called *native mode* or *block I/O* files. On RSX systems, these files are called *files-11* files.

Native mode and files-11 format files have a simpler file structure than RMS–11 files and are stored as a stream of ASCII characters with embedded line feeds and carriage returns indicating the end of the records. The simpler structure makes these files easier to access, although they cannot perform the complex functions RMS–11 files can perform. See the *RSTS/E Programmer's Utilities Manual* for information on RSTS/E native mode or block I/O files. See the *RSX–11M/M–PLUS Utilities Manual* for information on RSX files-11 format files.

The following sections describe how to use RMS files in BASIC–PLUS–2.

12.2 Record Formats

The format of a record determines how RMS stores the record in a block. You specify the record format in an OPEN statement. The following are valid BASIC–PLUS–2 record formats:

- Fixed-length records
- Variable-length records
- Stream records

12.2.1 Fixed-Length Records

Fixed-length records are all the same length. RMS stores fixed-length records as they appear in the record buffer, including any spaces or null characters following the data; this process is called *padding*. Processing these records involves less overhead than other record formats; however, this format can use disk storage space less efficiently than variable-length or stream records.

12.2.2 Variable-Length Records

Variable-length records can have different lengths, but no record can exceed a maximum size set for the file. When the record is written to a file, RMS adds a record length header that contains the length of the record (excluding the header) in bytes. When your program retrieves a record, this header is not included in the record buffer. While variable-length records usually make more efficient use of storage space than fixed-length records, manipulation of the record length headers generates processor overhead.

12.2.3 Stream Records

BASIC–PLUS–2 interprets stream records as a continuous sequence, or stream, of bytes. Unlike the fixed- and variable-length formats, stream records do not contain control information such as record counts, segment flags, or other system-supplied boundaries. Stream records are delimited by special characters or character sequences called *terminators*. In BASIC–PLUS–2, stream records can be delimited by any special character (usually a carriage return/line-feed pair). Note that stream record formats are valid only in sequential files.

12.3 File Organizations

BASIC–PLUS–2 provides several types of file organization: sequential, relative, indexed, and virtual. If you do not specify a file organization when creating a file, the default is a terminal-format file. The following sections describe each type of file organization.

Note

On RSTS/E systems, only files opened with ORGANIZATION SEQUENTIAL, RELATIVE, or INDEXED are RMS files. Files opened with ORGANIZATION VIRTUAL or with no ORGANIZATION clause are RSTS/E native-mode files.

12.3.1 Terminal-Format Files

On RSX systems, a terminal-format file is a sequential file of variable-length records. Terminal-format files are the default; that is, you create a terminal-format file when you do not specify a file organization when opening a file. You can then use the PRINT, INPUT, INPUT LINE, and LINPUT statements to access a terminal-format file. See Chapters 5 and 10 for more information about terminal-format files.

On RSTS/E systems, terminal-format files are RSTS/E native mode stream files. See Chapter 17 for more information about RSTS/E terminal-format files.

12.3.2 Sequential Files

A sequential file contains records that are stored in the order they are written. Sequential files can contain records of any valid BASIC–PLUS–2 record format: fixed-length, variable-length, or stream. Sequential files can reside on both disk and magnetic tape devices. Those stored on disk support shared access.

12.3.3 Relative Files

A relative file contains a series of "cells" that are numbered consecutively from 1 to *n*, where *n* represents the relative record number. Each cell can contain only a single record. For fixed-length records, the length of each cell equals the record length plus 1 byte. For variable-length records, the length of the cell equals the maximum record size plus 3 bytes.

You can access records in a relative file either sequentially or randomly. The relative record number is the *key value* in random access mode; that is, to access a record in a relative file in random access mode, you must know the relative record number of that record. You can add records to a relative file either at the end of the file or into any empty cell.

Relative files are most useful when randomly accessed and when the record can be identified by its cell number (for example, when inventory numbers correspond to cell numbers). Relative files support shared access. You can delete records from relative files, but not from sequential files.

12.3.4 Indexed Files

An indexed file contains data records that are sorted in ascending order according to a *primary index key value* (descending keys are not supported). The index key is a record field (or set of fields) that determines the order in which the records are logically accessed. Keys must be variables declared in a MAP statement. Keys can be either strings or WORD integers.

String keys can also be segmented; the key can be composed of up to eight string variables in a map.

Along with the primary index key value, you can also specify up to 254 alternate keys; RMS creates one index for each key you specify. For each of these keys you can also specify an ascending collating sequence. Each index is stored as part of the file, and each entry in the index contains a pointer to a record. Therefore, each key you specify corresponds to a sorted list of record pointers.

An indexed file of library books, for example, might be ordered by book title; that is, the title of the book is the primary key for the file. The keys for alternate indexes might include the author's name and the book's Library of Congress number. Neither of these alternate indexes contains the actual records; instead, these indexes contain sorted pointers to the appropriate records.

Indexed files are most useful when randomly accessed or when you want to access the records in more than one way.

12.3.5 Virtual Files

A virtual file is a random access file that stores one or more data records or virtual array elements in each physical 512-byte disk block. You create a virtual file by specifying the ORGANIZATION VIRTUAL clause as part of the OPEN statement and dimensioning an array on the file with the DIMENSION statement. You perform operations on virtual arrays just as you do with arrays in memory. Virtual array files allow you to use disk files for arrays too large to fit in memory.

12.4 Record Access and Record Context

Record access modes determine the order in which your program retrieves or stores records in a file. They determine the *record context*: the *current record* and the *next record* to be processed. When your program successfully executes any record operation, the current record and next record pointers can change. If a record operation is unsuccessful, these pointers do not change.

The four record access modes valid for RMS are as follows:

- Sequential access—valid on any file organization
- Random-by-record number access—valid on sequential fixed and all relative files
- Random-by-key access—valid on indexed files
- Random-by-RFA (Record File Address) access—valid on any RMS file located on disk

With sequential access, the next record is the next logical record in the file. In the case of relative files, the next logical record is the next existing record (deleted or never-written records are skipped). In the case of indexed files, the next logical record is the record with the next value in the current key of reference depending on that key's collating sequence. You can therefore access relative or indexed files sequentially by not specifying a relative record number or key value.

You can also access sequential fixed-length and relative files randomly by record number; that is, you can specify the record number of the record to be retrieved. For relative files, this record number corresponds to the cell number of the desired record.

You can access indexed files randomly by key. The key specification includes a primary or alternate key and its value. BASIC–PLUS–2 retrieves the record corresponding to that value in the particular key chosen.

You can access disk files of any organization by Record File Address (RFA); this means that you specify an RFA variable whose value uniquely identifies a particular record. The RFA requires six bytes of information. For more information about RFAs, see Section 12.7.9.

12.5 I/O and Record Buffers

An I/O buffer is a storage area in your program that RMS uses to store data for I/O operations. You do not have direct access to I/O buffers; they are controlled entirely by RMS. The I/O buffer holds blocks of data transferred from the device, and its size is always greater than or equal to that of the record buffer. For more information about the amount of storage allocated for I/O buffers, see the RMS documentation for your system.

A record buffer is another storage area in your program. You have direct access to and control of the record buffer. When your program reads a record from a file, the information is transferred from the file to the I/O buffer in one large chunk of data, and then the requested record is transferred to the record buffer. When your program writes a record, data is transferred from the record buffer to the I/O buffer, and then to the file either when the I/O buffer is full or when other blocks need to be read in.

You can use MAP statements to create *static* record buffers and associate program variables with areas (fields) of the buffer. Static record buffers are buffers whose size does not change during program execution and whose program variables are always associated with the same fields in the buffer.

You can create *dynamic* record buffers with the MAP DYNAMIC and REMAP statements. These statements, when used after a MAP statement, associate a particular program variable with a different area (field) of the record buffer. However, the total size of a record buffer does not change during program execution.

Note

If you do not specify a map, you must use MOVE TO and MOVE FROM statements to transfer data back and forth from the record buffer to program variables. However, MOVE statements do not transfer data to or from a file.

12.6 Accessing the Contents of a Record

BASIC–PLUS–2 provides several different methods for accessing the contents of a record:

- MAP statement
- MAP DYNAMIC, and REMAP statements (dynamic mapping)
- MOVE statements
- FIELD statements

The FIELD statement is a declining feature and is not recommended for new program development. It is recommended that you use either MAP statements, dynamic mapping, or MOVE statements to access record contents.

12.6.1 The MAP Statement

Normally, a record is divided into predetermined fields, the sizes of which are known at compile time. The MAP statement creates the storage area for this record and determines its total size.

Example 1

```
50     STRING last_name = 15,    &
              street_name = 30,  &
       INTEGER house_num

    MAP (People) name_addr student_info
```

Example 2

```
100  MAP (Emprec)
         STRING Emp_name = 25,                 &
         LONG Badge,                           &
         STRING Address = 25,                  &
         STRING Department = 4
```

12.6.2 The MAP DYNAMIC and REMAP Statements

There are situations where predetermined fields are not applicable or possible. In these situations, you must perform record defielding in your program at run time. Using the MAP DYNAMIC statement, you can specify the variables in the map whose positions can change at run time. The REMAP statement then specifies the new positions of the variables named in the MAP DYNAMIC statement.

The following example shows how you can use MAP, MAP DYNAMIC, and REMAP to deblock your record fields. The MAP statement allocates a storage area of 2048 bytes and names it *Emprec*. The MAP DYNAMIC statement specifies that the variables *Emp_name*, *Badge*, *Address*, and *Department* are

all located in *Emprec*, and that their positions can be changed at run time with the REMAP statement. The REMAP statement then redefines these variables to their appropriate sizes.

Example

```
100  MAP (Emprec) FILL$ = 2048
    MAP DYNAMIC (Emprec)                                    &
        STRING Emp_name,                                    &
        LONG Badge,                                         &
        STRING Address,                                     &
        STRING Department

    REMAP (Emprec) FILL$ = Record_offset,                   &
        Emp_name = 25,                                      &
        Badge,                                              &
        Address = 25,                                       &
        Department = 4
```

Note that when accessing virtual or sequential files, you can specify a RECORD clause for the GET statement. The following program opens a virtual file with each block containing 512 bytes. However, each block contains 4 logical records that are 128 bytes long. Each of these logical records consists of a 20-character first name field, a 44-character last name field, and a 64-character company name field.

Example

```
100     DECLARE WORD Record_number
        MAP (Virt) STRING FILL = 512
        MAP DYNAMIC (Virt) STRING First_name,        &
                              Last_name,             &
                              Company
        OPEN "VIRT.DAT" FOR INPUT AS FILE #5,        &
                     VIRTUAL, MAP Virt
        Record_number = 1%
        ON ERROR GOTO 19000
          WHILE -1%
             GET #5, RECORD Record_number
             FOR I% = 0% TO 3%
                REMAP (Virt) STRING FILL = (I% * 128%),  &
                      First_name = 20,                   &
                      Last_name = 44,                    &
                      Company = 64
               PRINT First_name, Last_name, Company
            NEXT I%
            Record_number = Record_number + 1%
         NEXT
```

```
19000   IF ERR = 11%
             THEN
                PRINT "Finished"
             RESUME 32767
             ELSE ON ERROR GOTO 0
             END IF
        END
```

After the first 512-byte block is brought into memory, the FOR...NEXT loop deblocks the data into 128-byte logical records. At each iteration of the FOR...NEXT loop, the REMAP statement uses the loop variable to mask off 128-byte sections of the block.

For more information on the MAP DYNAMIC and REMAP statements, see Chapter 7 and the *BASIC–PLUS–2 Reference Manual*.

12.6.3 The MOVE Statement

The MOVE statement defines data fields and moves them to and from the record buffer created by BASIC–PLUS–2. For example:

Example

```
MOVE FROM #9%, A$, Cost, Name$ = 30%, ID_num%
```

This statement moves a record with four data fields from the record buffer to the variables in the list:

- A string field *A$* with a default length of 16 characters
- A number field *Cost* of the default data type
- A second 30-character string field *Name$*
- An integer field *ID_num%*

Valid variables in the MOVE list are as follows:

- Scalar variables
- Arrays
- Array elements
- FILL items

Because BASIC–PLUS–2 dynamically assigns space for string variables, the default string length during a MOVE TO operation is the length of the string. The default for MOVE FROM is 16 characters. An entire array specified in a MOVE statement must include the array name, followed by n–1 commas, where n is the number of dimensions in the array. Note that these commas must be enclosed in parentheses.

You specify a single array element by naming the array and the subscripts of that element. The following statement moves three arrays from the program to the record buffer. *A$* specifies a one-dimensional string array, *C* specifies a two-dimensional array of the default data type, and *D%* specifies a three-dimensional integer array. *B*(3,2) specifies the element of array *B* that appears in row 3, column 2.

```
MOVE TO #5%, A$(), C(,), D%(,,), B(3,2)
```

Successive MOVE statements to or from the buffer start at the beginning of the record buffer. If a MOVE TO operation only partially fills the buffer, the rest of the buffer is unchanged. You use the GET statement to read a record from a file, and then you move the data from the buffer to variables and reference the variables in your program. A MOVE TO operation moves data from the variables into the buffer created by BASIC–PLUS–2. A PUT or UPDATE statement then moves the data from the buffer to the file.

The following program opens file MOV.DAT, reads the first record into the buffer, and moves the data from the buffer into the variables specified in the MOVE FROM statement.

Example

```
10  DECLARE STRING Emp_name, Address, Department
    DECLARE LONG Badge
    OPEN "MOV.DAT" AS FILE #1%,               &
         RELATIVE VARIABLE,                   &
         ACCESS MODIFY, ALLOW NONE,           &
         RECORDSIZE 512%
    GET #1%
    MOVE FROM #1%,                            &
         Emp_name = 25,                       &
         Badge,                               &
         Address = 25,                        &
         Department = 4

   .
   .
   .
    MOVE TO #1%,                               &
         Emp_name = 25,                        &
         Badge,                                &
         Address = 25,                         &
         Department = 4

    UPDATE #1%
    CLOSE #1%
    END
```

The MOVE TO statement moves the data from the named variables into the buffer. The UPDATE statement writes the record back into file MOV.DAT. The CLOSE statement closes the file.

12.7 File and Record Operations

You can perform a variety of operations on files and on the records within a file. The following is a list of all the file and record operations supported by BASIC–PLUS–2:

- Open a file for processing with the OPEN statement.
- Locate a record in a file with the FIND statement.
- Read a record from a file with the GET statement.
- Write a record to a file with the PUT statement.
- Delete a record from a file with the DELETE statement.
- Change the contents of a record field with the UPDATE statement.
- Unlock the last record accessed with the UNLOCK statement.
- Unlock all previously locked records with the FREE statement.
- Write data to a terminal-format file with the PRINT # statement.
- Read data from a terminal-format file with the INPUT # statement.
- Reset the current record pointer to the beginning of a file with the RESTORE/RESET # statements.
- Delete all the records after a certain point; that is, truncate the records, with the SCRATCH statement.
- Rename a file with the NAME AS statement.
- Close an open file with the CLOSE statement.
- Delete an entire file with the KILL statement.

Note that before you can perform any operations on the records in a file, you must first open the file for processing.

12.7.1 Opening Files

The OPEN statement opens a file for processing, specifies the characteristics of the file to RMS, and verifies the result. Opening a file with the specification FOR INPUT specifies that you want to use an existing file. Opening a file with the specification FOR OUTPUT indicates that you want to create a new file. If you do not specify FOR INPUT or FOR OUTPUT, BASIC–PLUS–2 tries to open an existing file. If no such file exists, BASIC–PLUS–2 then creates a new file. Note that if you do not specify an ACCESS clause, files are opened with MODIFY ACCESS by default.

Clauses to the OPEN statement allow you to specify the characteristics of a file. All OPEN statement clauses concerning file or record format are optional when you open an existing file; those attributes that are not specified default to the attributes of the existing file. When you open an existing file, you must specify the file name, channel number, and unless the file is a terminal-format file, an organization clause. If you do not know the organization of the file you want to open, you can specify ORGANIZATION UNDEFINED. If you specify ORGANIZATION UNDEFINED, also specify RECORDTYPE ANY.

If you do not specify a map in the OPEN statement, the size of your program's record buffer is determined by the OPEN statement RECORDSIZE clause, or by the record size associated with the file. If you specify both a MAP clause and a RECORDSIZE clause in the OPEN statement, the specified record size overrides the size specified by the MAP clause.

The following statement opens a new sequential file of stream format records:

```
OPEN "TEST.DAT" FOR OUTPUT AS FILE #1%,        &
      SEQUENTIAL STREAM
```

The following example creates a relative file and associates it with a static record buffer. The MAP statement defines the record buffer's total size and the data types of its variables. When the program is compiled, BASIC–PLUS–2 allocates space in the record buffer for one integer, one 16-byte string, and one double-precision floating-point number. The record size is the total of these fields, or 28 bytes. All subsequent record operations use this static buffer for I/O to the file.

Example

```
10  MAP (Inv) LONG Part_number,                        &
              STRING Inv_name = 16,                    &
              DOUBLE Unit_price
    OPEN "INVENT.DAT" FOR OUTPUT AS FILE #1%           &
              ,ORGANIZATION RELATIVE FIXED, ACCESS MODIFY  &
              ,ALLOW READ, MAP Inv
```

The following OPEN statement opens a sequential file for reading only (ACCESS READ). Because the OPEN statement does not contain a MAP clause, BASIC–PLUS–2 creates a record buffer. This record buffer is 100 bytes long.

Example

```
10  OPEN "CASE.DAT" AS FILE #1%                     &
            ,ORGANIZATION SEQUENTIAL VARIABLE       &
            ,ACCESS READ                            &
            ,RECORDSIZE 100%
```

When you do not specify a MAP statement, your program must use MOVE TO and MOVE FROM statements to move data between the record buffer and a list of variables.

The OPEN statement for indexed files must have a MAP clause. Moreover, if you are creating an indexed file, a PRIMARY KEY clause is required. You can create a segmented index key containing more than one string variable by separating the variables with commas and enclosing them in parentheses. All the string variables must be part of the associated map. In the following example, the primary key is made up of three string variables. This key causes the records to be sorted in alphabetical order according to the person's last name, first name, and middle initial.

Example

```
10  MAP (Segkey) STRING First_name = 15, MI = 1, Last_name = 15
    OPEN "NAMES.IND" FOR OUTPUT AS FILE #1%,                  &
          ORGANIZATION INDEXED,                               &
          PRIMARY KEY (Last_name, First_name, MI),            &
          MAP Segkey
```

Note that there are restrictions on the maximum record size allowed for various file and record formats. For more information, see the RMS documentation for your system.

You can identify a device either by a physical device name or by a logical device name. Logical names are assigned using the DCL command ASSIGN. An example of the ASSIGN command is as follows:

```
$ ASSIGN DU1: MYDISK:
```

This command assigns the logical name MYDISK: to the disk DU1:.

You can use the DEASSIGN command to delete logical name assignments. The following DEASSIGN command deletes the logical name MYDISK:.

```
$ DEASSIGN MYDISK:
```

The main advantage in using logical names is that your programs no longer depend on the availability of a specific device. In the following example, for instance, the magnetic tape drive MM2: must be available for your program to run. If it is not available, you must recode the OPEN statement to specify another device, recompile, build, and then link your program.

```
10   OPEN 'MM2:TEST.FIL' FOR OUTPUT AS FILE #2%
```

If you change the OPEN statement to include a logical name such as MYTAPE:, however, you can assign any available device to MYTAPE:. The new OPEN statement looks like the following:

```
10   OPEN 'MYTAPE:TEST.FIL' FOR OUTPUT AS FILE #2%
```

When MM2: is not available, you can move your magnetic tape to a new magnetic tape drive and assign the logical name MYTAPE: to the new device.

See the *RSTS/E System User's Guide* or the *RSX-11M-PLUS Command Language Manual* for more information on the ASSIGN and DEASSIGN commands.

12.7.2 Creating Virtual Array Files

BASIC-PLUS-2 virtual arrays let you define arrays that reside on disk. You use them just as you would an ordinary array. You create a virtual array by dimensioning an array with the DIM # statement, then opening a VIRTUAL file on that channel. You access virtual arrays just as you do normal arrays. The following DIM # statement dimensions a virtual array on channel #1. The OPEN statement opens a virtual file that contains the array. The last program line assigns a value to one array element.

Example

```
100  DIM #1%, LONG Int_array(10,10,10)
       .
       .
       .
     OPEN "VIRT.DAT" FOR OUTPUT AS FILE #1%, VIRTUAL
       .
       .
       .
     Int_array(5,5,5) = 100%
```

Note that you cannot redimension virtual arrays with an executable DIM statement. See Chapter 10 for more information on virtual arrays.

12.7.3 Locating Records

The FIND statement locates a specified record and makes it the current record. Using the FIND statement to locate records can be faster than using a GET statement because the FIND statement does not transfer any data to the record buffer; therefore, it executes faster than a GET statement. However, if you are interested in the contents of a record, you must retrieve it with a GET operation.

The FIND statement sets the current record pointer to the record just found, making it the target for a GET, UPDATE, or DELETE statement. (Note that you must have write access to a record before issuing a PUT, UPDATE, or DELETE operation.) A sequential FIND operation searches records in the following order:

- Sequential files from beginning to end
- Relative files in ascending relative record or cell number order
- Indexed files in ascending order, based on the current key of reference and the key's collating sequence

For sequential fixed-length and relative files, you can find a particular record by specifying a RECORD clause. This is called a *random access FIND*. You can also perform a random access FIND for indexed files by specifying a key of reference, a relational test, and a key value. In the following example, the first FIND statement finds the first record whose key value either equals or follows SMITH in the key's collating sequence. The second FIND statement finds the first record whose key value follows JONES in the key's collating sequence. Each record found by the FIND statement becomes the current record. (Note that you can only have one current record at a time.)

Example

```
10  MAP (Emp) STRING Emp_name, WORD Emp_number, SSN
    OPEN "EMP.DAT" AS FILE #1%, INDEXED,          &
             ACCESS READ,                          &
             MAP Emp,                              &
             PRIMARY KEY Emp_name
    FIND #1%, KEY #0% GE "SMITH"
    FIND #1%, KEY #0% GT "JONES"
```

The string expression can contain fewer characters than the key of the record you want to find. However, if you want to locate a record whose string key field *exactly* matches the string expression you provide, you must pad the string expression with spaces to the exact length of the key of reference. For example:

Example

```
10  FIND #5%, KEY #0% EQ "TOM     "
    FIND #5%, KEY #0% EQ "TOM"
```

The first FIND statement locates a record whose primary key field equals "TOM   ". The second FIND statement locates the first record whose primary key field begins with "TOM".

Table 12–1 displays the status of the current record and next record pointers after both a sequential and a random access FIND.

Table 12–1 Record Context After a FIND Operation

Record Access Mode	File Type	Current Record	Next Record
Sequential FIND	Sequential	Record found	Current record + 1
	Relative	Record found	Next existing record
	Indexed	Record found	Next record in current key order
Random access FIND	All	Record found	Unchanged

Note that a random access FIND operation locates the specified record and makes it the current record, but the next record pointer does not change.

12.7.4 Reading Records

The GET statement moves a record from a file to a record buffer and makes the data available for processing. GET statements are valid on sequential, relative, and indexed files. You should not use GET statements on terminal-format files, or virtual array files.

For sequential files, a sequential GET retrieves the next record in the file. For relative files, a sequential GET retrieves the next existing record. For indexed files, a sequential GET retrieves the record with the next ascending value in the current key of reference, depending on that key's collating sequence.

Table 12–2 shows the current record and next record pointers after a GET operation. Note that the values of these pointers vary depending on whether or not the previous operation was a FIND.

Table 12–2 Record Context After a GET Operation

Record Access Mode	File Type	Current Record	Next Record
Sequential GET with FIND	Sequential	Record found	Current record + 1
	Relative	Record found	Next existing record
	Indexed	Record found	Next record in current key
Sequential GET without FIND	Sequential	Next record	Next record + 1
	Relative	Next existing record	Next existing record + 1
	Indexed	Next record in current key	Record following next record in current key
Random GET	All	Record specified	Next record in succession

If you precede a sequential GET operation with a FIND operation, the current record is the one located by FIND. If you do not perform a FIND operation before a sequential GET operation, the current record is the next sequential record.

The following example illustrates the use of the GET operation to sequentially access records in an indexed file. The example opens an indexed file and displays the first 25 records with serial numbers greater than AB2721 in ascending primary key value order.

Example

```
10  MAP (Bec) STRING Owner = 30%, WORD Vehicle_number,     &
              STRING Serial_number = 22%
    OPEN "VEH.IDN" FOR INPUT AS FILE #2%,                  &
          ORGANIZATION INDEXED, PRIMARY KEY Serial_number, &
          MAP Bec, ACCESS READ
    GET #2%, KEY #0% EQ "AB2721"
    FOR I% = 1% TO 25%
          GET #2%
          PRINT "Vehicle Number = ";Vehicle_number
          PRINT "Owner is: ";Owner
          PRINT
    NEXT I%
```

The following example performs random GET operations by specifying a record number:

Example

```
10  MAP (Bec) STRING Owner = 30%, WORD Vehicle_number,     &
              STRING Serial_number = 22%
    OPEN "VEH.IDN" FOR INPUT AS FILE #2%,                  &
          ORGANIZATION SEQUENTIAL FIXED,                   &
          MAP Bec, ACCESS READ
    INPUT "Which record do you want";A%
    WHILE (A% <> 0%)
       GET #2%, RECORD A%
       PRINT "The vehicle number is", Vehicle_number
       PRINT "The serial number is", Serial_number
       PRINT "The owner of vehicle";Vehicle_number; "is", Owner
       INPUT "Next Record";A%
    NEXT
    CLOSE #2%
    END
```

If you are trying to access a locked record, BASIC–PLUS–2 signals "Record/bucket locked" (ERR=154).

12.7.5 Writing Records

For a file opened with ACCESS WRITE or ACCESS MODIFY, the PUT statement moves data from the record buffer to a file using the I/O buffer. PUT statements are valid on RMS sequential, relative, and indexed files. You cannot use PUT statements on terminal-format files, or virtual array files.

Sequential access is valid on RMS sequential, relative, and indexed files. For sequential, variable, and stream files, a sequential PUT operation adds a record at the end of the file. For sequential fixed and relative files, PUT writes records sequentially or randomly depending on the presence of a RECORD clause. For indexed files, RMS stores records in order of the primary key's collating sequence. Therefore, you do not need to specify a random or sequential PUT. The following table shows the record context after both random and sequential PUT operations.

Table 12–3 Record Context After a PUT Operation

Record Access Mode	File Type	Current Record	Next Record
Sequential PUT	Sequential	None	End of file
Sequential PUT	Relative	None	Next record
Sequential PUT	Indexed	None	Undefined
Random PUT	Relative	None	Unchanged

After a PUT operation, the current record pointer has no value. However, the value of the next record pointer changes depending on the file type and the record access mode used with the PUT operation. In a sequential, stream, or variable file, records can be added only at the end of the file; therefore, the next record after PUT is the end of the file. In a relative, sequential, or fixed file, the next record after a PUT operation is the next logical record.

The following example opens a sequential file with ACCESS APPEND specified. For sequential files, this is almost identical to ACCESS WRITE. The only difference is that, with ACCESS APPEND, BASIC–PLUS–2 positions the file pointer after the last record in the file when it opens the file for processing. All subsequent PUT operations append the new record to the end of the existing file.

Example

```
50  MAP (Buff) STRING Code = 4%, Exp_date = 9%, Type_desig = 32%
    OPEN "INV.DAT" FOR INPUT AS FILE #2%,                      &
          ORGANIZATION SEQUENTIAL FIXED, ACCESS APPEND,        &
          MAP Buff
    WHILE -1%
          INPUT "What is the specification code";Code
          INPUT "What is the expiration date";Exp_date
          INPUT "What  is the designator";Type_desig
          PUT #2%
    NEXT
```

If the current record pointer is not at the end of the file when you attempt a sequential PUT operation to a sequential file, BASIC–PLUS–2 signals “Not at end of file” (ERR=149).

In the following example, the PUT statement writes records to an existing indexed file. In this case, the error message “Duplicate key detected” (ERR=134) indicates that a record with a matching key field already exists, and you did not allow duplicates on that key.

Example

```
10      MAP (Myrec) STRING R_num = 5,                          &
                                Dept_name = 10,               &
                                Pur_dat = 9
20      OPEN "INFO.DAT" FOR OUTPUT AS FILE #2,                   &
                ORGANIZATION INDEXED FIXED, ACCESS WRITE, &
                PRIMARY KEY R_num, MAP Myrec
30      WHILE -1%
                INPUT "Requisition number";R_num
                INPUT "Department name";Dept_name
                INPUT "Date of purchase";Pur_dat
                PRINT
                PUT #2%
        NEXT
```

Output

```
Requisition number? 2522A
Department name? COSMETICS
Date of purchase? 15-JUNE-1990

Requisition number? 2678D
Department name? AUTOMOTIVE
Date of purchase? 15-JUNE-1990

Requisition number? 4167C
Department name? AUTOMOTIVE
Date of purchase? 6-JANUARY-1990

Requisition number? 2522A
Department name? SPORTING GOODS
Date of purchase? 25-FEBRUARY-1990

%Duplicate key detected at line   30 in "MAP "
```

12.7.6 Deleting Records

The DELETE statement removes a record from a file that was opened with ACCESS MODIFY. After you have deleted a record you cannot retrieve it. DELETE works with relative and indexed files only.

A successful FIND or GET operation must precede the DELETE operation. These operations make the target record available for deletion. In the following example, the FIND statement locates record 67 in a relative file, and the DELETE statement removes this record from the file. Because the cell itself is not deleted, you can use the PUT statement to write a record into that cell after deleting its contents.

Example

```
10  FIND #1%, RECORD 67%
    DELETE #1%
```

Note

There is no current record after a deletion. The next record pointer is unchanged.

12.7.7 Updating Records

UPDATE writes a new record at the location indicated by the current record pointer. UPDATE is valid on RMS sequential, relative, and indexed files.

The UPDATE statement operates on the current record, provided that you have write access to that record. In order to successfully update a variable-length record, you must know the exact size of the record you want to update. BASIC–PLUS–2 has access to this information after a successful GET operation. If you have not performed a successful GET operation on the variable-length record, then you must specify a COUNT clause in the UPDATE statement that contains the record size information.

An UPDATE will fail with the exception "No current record" (ERR=131) if you have not previously established a current record with a successful GET or FIND. Therefore, when updating records you should include error trapping in your program to make sure all GET operations execute successfully.

An UPDATE operation on a sequential file is valid only when:

- The file containing the record is on disk.
- The new record is the same size as the one it is replacing.
- You have established a current record via a GET or FIND operation. Note that COUNT defaults to the size of the current record if a GET was performed. If a FIND operation was used to locate the current record, then you must supply a COUNT value.

The following program searches to find a record in which the *L_name* field matches the specified *Search_name$*. Once this record is found and retrieved, the *Rm_num* field of that record is updated; the program then prompts for another *Search_name$*. If a match is not found, BASIC–PLUS–2 prints the message "No such record" and prompts the user for another *Search_name$*. The program ends when the user enters a null string for the *Search_name$* value.

Example

```
20      MAP (AAA) STRING L_name = 60%, F_name = 20%, Rm_num = 8%
30      OPEN "STU.DAT" FOR INPUT AS FILE #9%,           &
                ORGANIZATION SEQUENTIAL FIXED, MAP AAA
50      INPUT "Last name";Search_name$
55      Search_name$ = EDIT$(Search_name$, -1%)
60      IF Search_name$ = ""
                THEN GOTO 32010
        END IF
65      RESTORE #9%
70      ON ERROR GOTO 19000
75         GET #9% WHILE Search_name$ <> L_name
80      INPUT "Room number"Rm_num
90      UPDATE #9%
100     GOTO 50
   .
   .
   .
19000   IF ERR=11
              THEN
                  PRINT "No such record"
                  RESUME 50
              ELSE
                  ON ERROR GOTO 0
           END IF
32010   CLOSE #9%
32030   PRINT "Update complete"
32767   END
```

Note

An UPDATE operation invalidates the value of the current record pointer. The next record pointer is unchanged.

When you update a record in a relative variable file, the new record can be larger or smaller than the record it replaces, provided that it is smaller than the maximum record size set for the file. When you update a record in an indexed variable file, the new record can also be larger or smaller than the record it replaces. The updated record:

- Can be no longer than the maximum record size, if specified
- Must include at least the primary key field

The following program updates a specified record on an indexed file:

Example

```
200  MAP (UPD) STRING Enrdat = 8%, LONG Part_num, Sh_num, REAL Cost
210  OPEN "REC.ING" FOR INPUT AS FILE #8%,                       &
          INDEXED, MAP UPD, PRIMARY KEY Part_num
300  INPUT "Part number to update";A%
500  Loop1:
     WHILE -1%
       GET #8%, KEY #0%, EQ A%
       INPUT "Revised Cost is";Cost
       UPDATE #8%
       INPUT "Next Record";A%
       IF A% = 0%
       THEN
            EXIT Loop1
       END IF
600  NEXT
700  CLOSE #8%
800  END
```

If the new record either omits one of the old record's alternate key fields or changes one of them, the OPEN statement must specify a CHANGES clause for that key field when the file is created. Otherwise, BASIC–PLUS–2 signals the error "Key not changeable" (ERR=130).

12.7.8 Controlling Record Access

When you open a file, BASIC–PLUS–2 allows you to specify how you will access the file and what types of access you will allow other running programs while you have the file open.

If you open a file for read access only (ACCESS READ), BASIC–PLUS–2 by default allows other programs to have unrestricted access to the file. You can restrict access with an ALLOW clause only if the file's security constraints allow you write access to the file.

BASIC–PLUS–2 by default prevents access by other programs to any file you open with ACCESS WRITE, ACCESS MODIFY, or ACCESS SCRATCH (sequential files only). This default action is equivalent to specifying the OPEN statement ALLOW NONE clause. To allow less restrictive access to the open file, specify ALLOW READ or ALLOW MODIFY.

When a file is open for read access only and you have not restricted access to other programs with ALLOW NONE, BASIC–PLUS–2 allows other programs to read any record in the file, including records that your program is concurrently accessing. However, when you retrieve a record with the GET statement from a file you have opened with the intent to modify, BASIC–PLUS–2 normally restricts other programs from accessing that record. This restriction is called *locking*.

To allow other programs to access a record you have locked, you must release the lock on the record in one of the following ways:

- Retrieve another record on the same channel.
- Explicitly unlock the record with the UNLOCK or FREE statement. The UNLOCK statement releases the current record. The FREE statement releases all records locked on a given channel.
- Perform an UPDATE operation on the record. An UPDATE statement causes the current record to be unlocked.
- Close the file.

In addition to the capability of restricting access via the OPEN statement ALLOW clause, BASIC–PLUS–2 allows programs to explicitly control record locking on each record that is retrieved.

12.7.9 Accessing Records by Record File Address

A Record File Address (RFA) uniquely specifies a record in a file. Accessing records by RFA is therefore more efficient and faster than other forms of random record access.

Because an RFA requires six bytes of storage, BASIC–PLUS–2 has a special data type, RFA, that denotes variables that contain RFA information. Variables of data type RFA can be used only with the I/O statements and functions that use RFA information, and in comparison and assignment statements. You cannot print these variables or use them in any arithmetic operation; however, you can compare RFA variables using the equal to (=) and not equal to (<>) relational operators.

You cannot create named constants of the RFA data type. However, you can assign values from one RFA variable to another, and you can use RFA variables as parameters.

Accessing a record by RFA requires three steps:

1. Explicitly declare the variable or array of data type RFA to hold the address.
2. Assign the address to the variable or array element. You can do this either with the GETRFA function, or by reading a file of RFAs generated by previous GETRFA functions.
3. Specify the variable in the RFA clause of a GET or FIND statement.

The GETRFA function returns the RFA of the last record accessed on a channel. Therefore, you must access a record in the file with a GET, FIND, or PUT statement before using the GETRFA function. Otherwise, GETRFA returns a zero, which is an invalid RFA. The following example declares an array of type RFA containing 100 elements. After each PUT operation, the RFA of the record is assigned to an element of the array. Once the RFA information is assigned to a program variable or array element, you can use the RFA clause on a GET or FIND statement to retrieve the record.

Example

```
100 DECLARE RFA R_array(100)
    DECLARE LONG I
    MAP (XYZ) STRING A = 80
    OPEN "TEST.DAT" FOR OUTPUT AS FILE #1,      &
          SEQUENTIAL, MAP XYZ
    FOR I = 1% TO 100%
  .
  .
  .
        PUT #1
        R_array(I) = GETRFA(1%)
    NEXT I
```

You can use the RFA clause on GET statements for any file organization; the only restriction is that the file must reside on a disk that is accessible to the node that is executing the program. The following example continues the previous one. It randomly retrieves the records in a sequential file by using RFAs stored in the array.

Example

```
200 DECLARE RFA R_array(100%)
    DECLARE LONG I
    MAP (XYZ) STRING A = 80
    OPEN "TEST.DAT" FOR OUTPUT AS FILE #1,      &
          SEQUENTIAL, MAP XYZ
    FOR I = 1% TO 100%
  .
  .
  .
        PUT #1
        R_array(I) = GETRFA(1%)
    NEXT I
    WHILE -1%
        PRINT "Which record would you like to see";
        INPUT "type a carriage return to exit";Rec_num%
        EXIT PROGRAM IF Rec_num% = 0%
        GET #1, RFA R_array(Rec_num%)
        PRINT A
    NEXT
```

12.7.10 Transferring Data to Terminal-Format Files

The PRINT # statement transfers program data to a terminal-format file. In the following example, the INPUT statements prompt the user for three values: *S_name$*, *Area$*, and *Quantity%*. Once these values are entered, the PRINT # statement writes these values to a terminal-format file that is open on channel #4.

Example

```
100  FOR I% = 1% TO 10%
        INPUT "Name of salesperson";S_name$
        INPUT "Sales district";Area$
        INPUT "Quantity sold";Quantity%
        PRINT #4%, S_name$, Area$, Quantity%
     NEXT I%
```

If you do not specify an output list in the PRINT # statement, a blank line is written to the terminal-format file. A PRINT statement without a channel number transfers program data to a terminal. See Chapter 5 for more information.

12.7.11 Resetting the File Position

The RESTORE # statement resets the current record pointer to the beginning of the file; it does not change the file. RESET # is a synonym for RESTORE. For example:

Example

```
100  RESTORE #3%, KEY #2%
     RESET #3%
```

The RESTORE # statement restores the file in terms of the second alternate key. The RESET # statement restores the file in terms of the primary key.

The RESTORE # statement can be used by all RMS file organizations. RESTORE without a channel number resets the data pointer for READ and DATA statements but does not affect any files.

12.7.12 Truncating Files

The SCRATCH statement is valid only on sequential files. Although you cannot delete individual records from a sequential file, you can delete all records starting with the current record through to the end of the file. In order to do this, you must first specify ACCESS SCRATCH when you open the file.

To truncate the file, locate the first record to be deleted. Once the current record pointer points to this record, execute the SCRATCH statement. The following program locates the thirty-third record and truncates the file beginning with that record.

Example

```
10  OPEN "MMM.DAT" AS FILE #2%,            &
          SEQUENTIAL FIXED, ACCESS SCRATCH
    first_bad_record = 33%
    FIND #2%, RECORD first_bad_record
    SCRATCH #2%
    CLOSE #2%
    END
```

SCRATCH does not change the physical size of the file; it reduces the amount of information contained in the file. Therefore, you can write records with the PUT statement immediately after a SCRATCH operation.

12.7.13 Renaming Files

If the security constraints permit, you can change the name or directory of a file with the NAME...AS statement. For example:

Example

```
20  NAME "MONEY.DAT" AS "ACCOUNT.DAT"
```

This statement changes the name of the file MONEY.DAT to ACCOUNTS.DAT.

Note

The NAME...AS statement can change only the name and directory of a file; it cannot be used to change the device name.

You must always include an output file type because there is no default. If you use the NAME...AS statement on an open file, the new name does not take effect until you close the file.

12.7.14 Closing Files and Ending I/O

All programs should close files before the program terminates. However, BASIC–PLUS–2 automatically closes files in the following situations:

- At an END, END PROGRAM, or EXIT PROGRAM statement
- When it completes the last statement in the program if no END statement exists
- While executing a CHAIN statement

BASIC–PLUS–2 does not close files after executing a STOP, END SUB, or END FUNCTION statement.

The CLOSE statement closes files and disassociates these files and their buffers from the channel numbers. If the file is a magnetic tape device and the data is written to a tape, CLOSE writes trailer labels at the end of the file. The following is an example of the CLOSE statement:

Example

```
300  CLOSE #1%
     B% = 4%
     CLOSE #2%, B%, 7%
     CLOSE I% FOR I% = 1% TO 20%
```

12.7.15 Deleting Files

If the security constraints permit, you can delete a file with the KILL statement.

```
100  KILL "TEST.DAT"
```

This statement deletes the file named TEST.DAT. Note that this statement deletes only the most current version of the file. Do not omit the file type, because there is no default. You can delete only one file at a time.

You can delete a file that is currently being accessed by other users; however, the file is not deleted until all users have closed it. You cannot open or access a file once you have deleted it.

12.8 File-Related Functions

BASIC–PLUS–2 provides built-in functions for finding the following:

- The characteristics of the last file opened (FSP$)
- The file name and status of the specified file name string (FSS$)
- The number of bytes moved in the last I/O operation (RECOUNT)
- The file status (STATUS)

These functions are discussed in the following sections.

12.8.1 The FSP$ Function

If you do not know the organization of a file, you can find out by opening the file for input with the ORGANIZATION UNDEFINED and RECORDTYPE ANY clauses. Your program can then use the FSP$ function to determine the characteristics of that file. Your program must execute FSP$ immediately after the OPEN FOR INPUT statement.

Example

```
10  MAP (A) A$ = 32
20  MAP (A) WORD ORG_RAT, MRS, LONG ALQ,         &
        WORD BKS_BLS, NUM_KEYS, LONG MRN         &
30  OPEN "FIL.DAT" FOR INPUT AS FILE  #1%,       &
        ORGANIZATION UNDEFINED,
        RECORDTYPE ANY, ACCESS READ
40  A$ = FSP$(1%)
```

In this example, FSP$ generates the following values:

- ORG_RAT, which returns file characteristics:
  - High Byte is the RMS organization (ORG) field.
  - Low Byte is the RMS record attributes (RAT) field.
- MRS returns the RMS maximum record size (MRS) field.
- ALQ returns the RMS allocation quantity (ALQ) field.
- BKS_BLS returns the RMS bucket size (BKS) field for disk files or the RMS block size (BLS) field for magnetic tape files.
- NUM_KEYS returns the number of keys.
- MRN returns the RMS maximum record number (MRN) if the file is relative.

For more information on the FSP$ function, see the *BASIC–PLUS–2 Reference Manual* and the RMS documentation for your system.

12.8.2 The FSS$ Function

The FSS$ function performs a file name scan on the argument string and returns a string describing the file name and status. Because file specifications differ from system to system, the returned string contains system-specific information.

The output returned by the FSS$ function is a 30-character string encoded as shown in Table 12–4.

Table 12–4 File Name String: Flag Word Bytes 1–30

Byte	Meaning
1	Job number multiplied by two.
2	Undefined on RSTS/E systems. On RSX systems, byte 2 contains the version number. If the version number is undefined, the byte returns zero.
3–4	Undefined on RSTS/E systems. On RSX systems, this byte contains the last three characters of the file name.
5–6	Project and programmer number.
7–10	File name in Radix–50 format.
11–12	File extension in Radix–50 format.
13–14	Undefined on RSX systems. On RSTS/E systems, these bytes contain the FILESIZE clause specification.
15–16	Undefined on RSX systems. On RSTS/E systems, these bytes contain the CLUSTERSIZE clause specification.
17–18	Undefined on RSX systems. On RSTS/E systems, these bytes contain the MODE clause specification.
19–20	Undefined.
21	Undefined on RSX systems. On RSTS/E systems, this byte contains zero unless a protection code is specified or a default exists.
22	Protection code if byte 21 is non-zero.
23–24	Device name if specified. If the device name is logical and not translatable, byte 23 contains the first two characters in ASCII format; byte 24 contains the last two characters.
25	Unit number of the device. If no device is given, byte 25 returns a zero.
26	255 if the unit number was specified.
27–28	Flag word 1. See Table 12–5.
29–30	Flag word 2. See Table 12–6.

Bytes 27 and 28 provide additional information about the file name as described in Table 12–5.

Table 12–5 File Name String: Scan Flag Word 1

Flag Word 1: Where S0% = M%(27%) + SWAP%(M%(28%))

Bit	Logical Test	Meaning
0	(S0% and 1%)<>0%	The /CLUSTERSIZE:n switch was specified.
	(S0% and 1%) = 0%	No /CLUSTERSIZE:n was found.
1	(S0% and 2%)<>0%	Either the /MODE:n or /RONLY switch was specified.
	(S0% and 2%) = 0%	Neither /MODE:n nor /RONLY was found.
2	(S0% and 4%)<>0%	Either the FILESIZE:n or /SIZE:n was specified.
	(S0% and 4%) = 0%	Neither the FILESIZE:n nor /SIZE:n was found.
3–7		Not used.
8	(S0% and 256%)<>0%	A file name was found in the source string (and is returned in Radix–50 format in bytes 7 through 10).
	(S0% and 256%) = 0%	No file name was found.
9	(S0% and 512%)<>0%	A dot (.) found in the source string.
	(S0% and 512%) = 0%	No dot was found in the source string, implying that no extension was specified.
10	(S0% and 1024%)<>0%	A project-programmer number was found in the source string.
	(S0% and 1024%) = 0%	No project-programmer number was found.
11	(S0% and 2048%)<>0%	A left angle bracket (<) was found in the source string, implying that a protection code was found.
	(S0% and 2048%) = 0%	No left angle bracket (<) was found, implying that no protection code was specified.
12	(S0% and 4096%)<>0%	A colon (:) (but not necessarliy a device name) was found.
	(S0% and 4096%) = 0%	No colon was found, implying that no device was specified.

(continued on next page)

Table 12–5 (Cont.) File Name String: Scan Flag Word 1

Flag Word 1: Where S0% = M%(27%) + SWAP%(M%(28%))		
Bit	**Logical Test**	**Meaning**
13	(S0% and 8192%)<>0%	A logical name was specified.
	(S0% and 8192%) = 0%	A device name was not specified.
15	S0%<0%	Source string contained wildcard characters in the file name, extension, or project-programmer fields. In addition, the specified device name does not correspond to any logical device assignments. The program must test bits of flag word 2 for wildcard characters and device name.

Bytes 29 and 30 provide additional information about the specified file name as described in Table 12–6.

Table 12–6 File Name String: Scan Flag Word 2

Flag Word 2: Where S1% = M%(29%) + SWAP%(M%(30%))		
Bit	**Logical Test**	**Meaning**
0	(S1% and 1%)<>0%	File name was found in the source string.
	(S1% and 1%) = 0%	No file name was found, and bits 1 and 2 return zero.
1	(S1% and 2%)<>0%	File name was an asterisk (*) and is returned in bytes 7 through 10 as the Radix–50 representation of the string "??????".
	(S1% and 2%) = 0%	File name was not an asterisk.
2	(S1% and 4%)<>0%	File name contained at least one question mark (?).
	(S1% and 4%) = 0%	File name did not contain any question marks.
3	(S1% and 8%)<>0%	A dot (.) was found.
	(S1% and 8%) = 0%	No dot was found, implying that no extension was specified and bits 4, 5, and 6 return zero.
4	(S1% and 16%)<>0%	An extension was found (the field after the dot was not null).
	(S1% and 16%) = 0%	No extension was found (the field after the dot was null), and bits 5 and 6 return zero.

(continued on next page)

Table 12–6 (Cont.) File Name String: Scan Flag Word 2

Flag Word 2: Where S1% = M%(29%) + SWAP%(M%(30%))

Bit	Logical Test	Meaning
5	(S1% and 32%)<>0%	The extension was an asterisk (*) and is returned in bytes 11 and 12 as the Radix–50 representation of the string "???".
	(S1% and 32%) = 0%	The extension was not an asterisk.
6	(S1% and 64%)<>0%	The extension contained at least one question mark (?).
	(S1% and 64%) = 0%	The extension did not contain any question marks.
7	(S1% and 128%)<>0%	A project-programmer number was found.
	(S1% and 128%) = 0%	No project-programmer number was found, and bits 8 and 9 return zero.
8	(S1% and 256%)<>0%	Project number was in the form of [*,PROG] and is returned in byte 6 as 255.
	(S1% and 256%) = 0%	Project number was not an asterisk.
9	(S1% and 512%)<>0%	Programmer number was in the form [PROJ,*] and is returned in byte 5 as 255.
	(S1% and 512%) = 0%	Programmer number was not an asterisk.
10	(S1% and 1024%)<>0%	A protection code was found.
	(S1% and 1024%) = 0%	No protection code was found.
11	(S1% and 2048%)<>0%	The protection code set as default by the current job was used.
	(S1% and 2048%) = 0%	The protection code is the default (<60>) or that found in the source string.
12	(S1% and 4096%)<>0%	A colon (but not necessarily a device name) was found.
	(S1% and 4096%) = 0%	No colon or device was found, and bits 13, 14, and 15 return zero.
13	(S1% and 8192%)<>0%	A device name was found.
	(S1% and 8192%) = 0%	No device name was found; bits 14 and 15 return zero.

(continued on next page)

Table 12–6 (Cont.) File Name String: Scan Flag Word 2

Flag Word 2: Where S1% = M%(29%) + SWAP%(M%(30%))		
Bit	**Logical Test**	**Meaning**
14	(S1% and 16384%)<>0%	A logical device name was specified.
	(S1% and 16384%) = 0%	An actual device name was specified; bit 15 returns zero.
15	S1%<0%	The specified device name was logical and is not assigned to an actual device. The logical name is returned in bytes 23 through 26 as a Radix–50 string.
	S1% = 0%	A physical device is assigned to the physical or logical device name. This physical device name is returned in bytes 23 and 24; unit information is returned in bytes 25 and 26.

The following is an example of the FSS$ function:

```
10      DIM Y%(30%)
20      LINPUT 'Enter file name string';A$
        INPUT 'Enter offset';B%
30      Y$ = FSS$ (A$, B%)
        CHANGE Y$ TO Y%
40      PRINT J%, Y%(J%) FOR J% = 1% TO 30%
32767   END

RUN
```

```
ENTER FILE NAME STRING? FILE.EXT
ENTER OFFSET? 1
1       32
2       0
3       0
4       0
5       1
6       1
7       244
8       38
9       64
10      31
11      20
12      35
13      0
14      0
15      0
16      0
17      0
18      0
19      0
20      0
21      0
22      0
23      68
24      66
25      0
26      255
27      0
28      23
29      153
30      48
```

In this program output:

1. Byte 1 (32) is the job number multiplied by 2.
2. Bytes 5 and 6 are the project and programmer number (1,1).
3. Bytes 7 through 12 are the file name and extension in Radix–50 format.
4. Bytes 23 and 24 are the device name in Radix–50 format.
5. Byte 26 indicates that the unit number was specified.
6. Byte 28 (23) is the binary number 00010111. The bits indicate the following:
   - The file name was found (bit 8).
   - A dot was found (bit 9).
   - The project and programmer numbers were found (bit 10).

- A colon was found (bit 12).

7. Byte 29 (153) is the binary number 010011001. The bits indicate the following:
   - The file name was found (bit zero).
   - A dot was found (bit 3).
   - The extension was found (bit 4).
   - The project and programmer numbers were found (bit 7).
8. Byte 30 (48) is the binary number 110000. The bits indicate the following:
   - A colon was found (bit 12).
   - The device name was found (bit 13).

12.8.3 The RECOUNT Function

Read operations can transfer varying amounts of data. The system variable RECOUNT contains the number of characters (bytes) read after each read operation.

After a read operation from your terminal, RECOUNT contains the number of characters transferred, including the line terminator. After accessing a record, RECOUNT contains the number of characters in the record.

RECOUNT is reset by every read operation on any channel, including the controlling terminal. Therefore, if you need to use the value of RECOUNT, copy it to another variable before executing another read operation. RECOUNT is undefined if an error occurs during a read operation.

RECOUNT is often used as the argument to the COUNT clause in the UPDATE or PUT statement for variable-length files. The following sequence of statements ensures that the output record on channel #5 is the same length as the input record on channel #4.

Example

```
80  GET #4%
    bytes_read% =  RECOUNT
    .
    .
    .
    PUT #5%, COUNT bytes_read%
```

12.8.4 The STATUS Function

The STATUS function returns an integer value containing information about the last opened channel. Your program can test each bit to determine the status of the channel.

The information returned depends on the error and is different on RSTS/E and RSX systems. For more information, see the *BASIC–PLUS–2 Reference Manual.*

12.9 OPEN Statement Clauses

This section describes the OPEN statement clauses that enable you to control how a file is created or opened. These clauses are as follows:

BUCKETSIZE
BUFFER
CONNECT
CONTIGUOUS
CLUSTERSIZE
DEFAULTNAME
EXTENDSIZE
FILESIZE
NOSPAN
RECORDTYPE
TEMPORARY
USEROPEN
WINDOWSIZE

For more information on the OPEN statement and OPEN statement clauses, see the *BASIC–PLUS–2 Reference Manual.*

12.9.1 The BUCKETSIZE Clause

The BUCKETSIZE clause applies only to relative and indexed files. A **bucket** is a logical storage structure that RMS uses to build and maintain relative and indexed files on disk devices. A bucket consists of 1 or more disk blocks. The default bucket size is the record size rounded up to a block boundary. Although RMS defines the bucket size in terms of disk blocks, the BUCKETSIZE clause specifies the number of records a bucket contains. For example:

Example

```
10  OPEN  "STOCK.DAT" FOR OUTPUT AS FILE #1%,          &
          ORGANIZATION RELATIVE FIXED, BUCKETSIZE 12%
```

This example specifies a bucket containing approximately 12 records. RMS reads the entire bucket into the I/O buffer at once, and a GET statement transfers one record from the I/O buffer to your program's record buffer.

When you open an existing relative or indexed file and specify a bucket size other than that originally assigned to the file, BASIC–PLUS–2 signals "File attributes not matched" (ERR=160).

Records cannot span bucket boundaries. Therefore, when you specify a bucket size in your program, you must consider the size of the largest record in the file. Note that a bucket must contain at least one record. Buckets in both relative and indexed files contain information in addition to the records stored in the bucket. You should take this into consideration.

There are two ways to establish the number of blocks in a bucket. The first is to use the BASIC–PLUS–2 default. The second is to specify the approximate number of records you want in each bucket. BASIC–PLUS–2 then calculates a bucket size based on that number.

The default bucket size assigned to relative and indexed files is as small as possible. A small bucket size, however, is rarely desirable.

BASIC–PLUS–2 selects a default bucket size depending on the following:

- The record length
- The file organization (relative or indexed)
- The record format

If you do not define the BUCKETSIZE clause in the OPEN statement, BASIC–PLUS–2 does the following:

- Assumes that there is a minimum of one record in the bucket
- Calculates a size
- Assigns the appropriate number of blocks

Note that when you specify a bucket size for files in your program, you must keep in mind the space versus speed trade-offs. A large bucket size increases file processing speed because a greater amount of data is available in memory at one time. However, it also increases the memory space needed for buffer allocation and the processing time required to search the bucket. Conversely, a small bucket size minimizes buffer requirements, but increases the number of accesses to the storage device, thereby decreasing the speed of operations.

12.9.2 The BUFFER Clause

The BUFFER keyword applies to disk files of any organization. In the case of sequential files, the BUFFER clause sets the number of blocks read in on each disk access. For relative and indexed files, the BUFFER clause determines the number of I/O buffers that are allocated. You can specify up to 255 buffers.

12.9.3 The CLUSTERSIZE Clause

The CLUSTERSIZE clause allows you to specify the smallest amount of contiguous disk space to be allocated when an RMS or RSTS/E file's present allocation is exhausted. The CLUSTERSIZE clause can be used on RSTS/E systems only. On RSX systems, use the EXTENDSIZE clause for similar functionality.

The integer expression you specify with the CLUSTERSIZE clause must be a power of two. For example, a CLUSTERSIZE of 8 means that each time the file requires more disk space, the RSTS/E operating system must have at least 8 contiguous blocks to allocate. If there is not enough contiguous space available, BASIC–PLUS–2 signals the error "No room for user on device" (ERR=4).

12.9.4 The CONNECT Clause

The CONNECT clause can be used only on indexed files. CONNECT lets you process different groups of records on different indexed keys or on the same key without incurring all of the RMS overhead of opening the same file more than once. For example, a program can read records in an indexed file sequentially by one key and randomly by another. Each stream is an independent, active series of record operations.

Example

```
10  MAP (Indmap) WORD Emp_num,                              &
         STRING Emp_last_name = 20,                         &
         SINGLE Salary,                                     &
         STRING Wage_code = 2
    OPEN "IND.DAT" FOR INPUT AS FILE #1%,                   &
         ORGANIZATION INDEXED,                              &
         MAP Indmap,                                        &
         PRIMARY KEY Emp_num,                               &
         ALTERNATE KEY Emp_last_name
   .
   .
   .
    OPEN "IND.DAT" FOR INPUT AS FILE #2%                    &
         ORGANIZATION INDEXED,                              &
         MAP Indmap,                                        &
         CONNECT 1
```

```
.
.
.
OPEN "IND.DAT" FOR INPUT AS FILE #3%          &
     ORGANIZATION INDEXED,                    &
     MAP Indmap,                              &
     PRIMARY KEY Emp_num,                     &
     ALTERNATE KEY Wage_code,                 &
     CONNECT 1
```

The channel on which you open the file for the first time is called the *parent*. The CONNECT clause specifies another channel on which you access the same file; connected channels are called *children*. More than one OPEN statement can connect to the parent channel; however, you cannot connect to a channel that has already been connected to another channel. If you close a parent file, all of its children must also be closed. See the *BASIC–PLUS–2 Reference Manual* for more information about the CONNECT clause.

12.9.5 The CONTIGUOUS Clause

A contiguous file with physically adjoining blocks minimizes disk searching and decreases file access time. Once the system knows where a contiguous file starts on the disk, it does not need to use as many retrieval pointers to locate the pieces of that file. Rather, it can access data by calculating the distance from the beginning of the file to the desired data. If there is not enough contiguous disk space, BASIC–PLUS–2 signals an error.

Opening a file with both the FILESIZE and CONTIGUOUS clauses preextends the file contiguously or in as few disk extents as possible.

12.9.6 The DEFAULTNAME Clause

The DEFAULTNAME clause in the OPEN statement lets you specify a default file specification for the file to be opened. On RSTS/E systems, you cannot specify the DEFAULTNAME clause with block I/O files; that is, files opened with ORGANIZATION VIRTUAL.

BASIC–PLUS–2 uses the DEFAULTNAME clause for any part of the file specification that is not explicitly supplied.

Example

```
20  LINPUT "Next data file";Fil$
30  OPEN Fil$ FOR INPUT AS FILE #5%,
         ORGANIZATION SEQUENTIAL,       &
         DEFAULTNAME "MYDISK:.DAT"
```

The DEFAULTNAME clause supplies default values for the device, directory, and file type portions of the file specification. Typing ABC in response to the "Next data file?" prompt causes BASIC–PLUS–2 to try to open MYDISK:ABC.DAT.

BASIC–PLUS–2 uses the DEFAULTNAME values only if you do not supply those parts of the file specification appearing in the DEFAULTNAME clause. For example, if you enter MYDISK:ABC in response to the prompt, BASIC–PLUS–2 tries to open MYDISK:ABC.DAT. In this case, MYDISK: overrides the device default in the DEFAULTNAME clause. Any part of the file specification still missing is filled in from the current default device and directory of the process.

12.9.7 The EXTENDSIZE Clause

The EXTENDSIZE attribute determines how many disk blocks RMS adds to the file when the current allocation is exhausted. The EXTENDSIZE clause only has an effect when creating a file. You specify EXTENDSIZE as a number of blocks. For example:

Example

```
10  OPEN "TSK.ORN" FOR OUTPUT AS FILE #2%,    &
          ORGANIZATION RELATIVE, EXTENDSIZE 128%
```

The EXTENDSIZE clause causes RMS to add 128 disk blocks whenever the current space allocation is exhausted and the file must be extended.

The value you specify must conform to the following requirements:

- It must be specified when you create the file.
- It cannot exceed 65,535 disk blocks.

If you specify zero, the extension size equals the default extension size for the volume. The EXTENDSIZE value can be overridden for single OPEN operations.

12.9.8 The FILESIZE Clause

With the FILESIZE attribute, you can allocate disk space for a file when you create it. The following statement allocates 50 blocks of disk space for the file "VALUES.DAT":

```
10  OPEN "VALUES.DAT" FOR OUTPUT AS FILE #3%, FILESIZE 50%
```

Preextending a file has several advantages:

- The system can create a complete directory structure for the file, instead of allocating and mapping additional disk blocks when needed.

- You reserve the needed disk space for your application. This ensures that you do not run out of space when the program is running.
- Preextension can make some of the file's disk blocks contiguous, especially when used with the CONTIGUOUS keyword.

Note that preextension can be a disadvantage if it allocates disk space needed by other users. The FILESIZE clause is ignored when BASIC–PLUS–2 opens an existing file.

12.9.9 The NOSPAN Clause

By default, sequential files allow records to cross or span block boundaries. If records cross block boundaries, RMS packs records into the file end-to-end throughout the file, leaving space for control information and padding.

The NOSPAN clause overrides this default, forcing records to fit into individual blocks (with space provided for control information and padding). When block boundaries restrict records, fixed-length records must be less than 512 bytes, and variable-length records less than 510 bytes. This can waste extra bytes at the end of each block. However, when records span block boundaries, RMS writes records end-to-end without regard for block boundaries. For example, if you specify NOSPAN, only four 120-byte records fit into a disk block. If you do not specify NOSPAN, BASIC–PLUS–2 begins writing the fifth record in the block, and continues writing that record in the next block. This minimizes wasted disk space and improves the file's capacity, at the minimal expense of increased processing overhead.

12.9.10 The RECORDTYPE Clause

The RECORDTYPE clause lets you specify record formats that are compatible with files created by other language processors. You can choose one of the following qualifiers:

- ANY
- FORTRAN
- LIST
- NONE

The default for BASIC–PLUS–2 is LIST, which specifies carriage return format. This is standard for ASCII text files and means that carriage control is performed by RMS when writing the file to a unit-record device.

If your program accesses a file created with a FORTRAN language processor, you may need to use the FORTRAN qualifier. In the following example, the FORTRAN qualifier sets the FORTRAN carriage control attribute in the RAT field in the FAB. For more information on the FAB control structure, see Section 12.9.12. The first byte of the record is assumed to be the carriage control information.

```
20  OPEN "FIL.DAT" FOR INPUT AS FILE #1%,         &
         ORGANIZATION SEQUENTIAL, RECORDTYPE FORTRAN
```

If your program accesses a file created by an unknown language processor or by DCL, the ANY qualifier causes BASIC–PLUS–2 to handle any record attribute type. Specifying the ANY qualifier is the same as specifying LIST.

```
20  OPEN "FIL.DAT" FOR INPUT AS FILE #1%,         &
         ORGANIZATION INDEXED, RECORDTYPE ANY
```

12.9.11 The TEMPORARY Clause

If you specify the TEMPORARY clause in the OPEN statement, BASIC–PLUS–2 deletes that file in any one of the following cases:

- When you close the file
- When the program aborts or exits
- When your process terminates

No entry for this file is made in any directory.

12.9.12 The USEROPEN Clause

The USEROPEN clause specifies a MACRO–11 routine that BASIC–PLUS–2 executes when you open or create a file. (You do not need to declare the USEROPEN routine with an EXTERNAL FUNCTION statement.) This procedure can then specify additional OPEN parameters for the file. For example:

Example

```
100  OPEN "FILE.DAT" FOR INPUT AS FILE #2%,       &
          ORGANIZATION INDEXED, USEROPEN Myopen, MAP ABC
```

The code in *Myopen* determines how the file FILE.DAT is opened.

A USEROPEN routine sets the File Access Block (FAB) and Record Access Block (RAB) RMS control structures. A USEROPEN procedure should not alter the allocation of these structures, although it can modify the contents of many of the fields. You should not modify fields set by other OPEN statement keywords. For example, you should use the RECORDSIZE clause, not a USEROPEN routine, to set the record length.

The allocation of the RMS control structures lasts only for the duration of the OPEN statement. Therefore, your USEROPEN routine can retain only the RAB address for use after the OPEN operation is complete. Note that any additional structures that you allocate and link into the RMS structures must be unlinked before exiting the USEROPEN.

The following steps describe the execution of the USEROPEN routine:

1. BASIC–PLUS–2 performs normal OPEN statement processing up to the point where it would call the RMS OPEN/CREATE and CONNECT routines. BASIC–PLUS–2 then passes control to the USEROPEN routine.
2. BASIC–PLUS–2 passes the address of the FAB as the first parameter and the address of the RAB as the second parameter.
3. The USEROPEN routine can modify the contents of the RMS control structures, and it must call the RMS OPEN or RMS CREATE routine and the RMS CONNECT routine and return the status in R0.

Note

The USEROPEN routine may use any register; however, the stack must be in the same state at routine exit as it was at routine entrance. RMS STS status value must be passed back to the OTS in R0.

You cannot use a USEROPEN routine to fill the RBF, UBF, BKS, or CTX fields in the RAB. These fields are filled in after the USEROPEN routine returns; any values placed there by the USEROPEN routine are overwritten.

Also, you must not set RMS Locate mode when using a USEROPEN routine on sequential files.

The following example uses a USEROPEN routine to set the protection of a file.

Example

```
        .TITLE      USR
;+
;       This routine will link a protection XAB to
;       the end of a linked list of XABs so that the
;       file will be created with a protection code different
;       from the default protection code for the disk it is on.
;
; INPUT:
;       2(R5) - Address to the FAB
;       4(R5) - Address to the RAB
;
; OUTPUT:
;       R0 - the STS field of either the FAB or RAB
;
; EFFECT:
;       The file is created a connect is done if no errors occured
;
;       R1 - R3 are destroyed.
;
; EXTERNALS:
        .MCALL      $GNCAL,FAB$B,RAB$B,XAB$B,NAM$B,$FBCAL,$RBCAL
        $GNCAL
        $FBCAL
        $RBCAL
;
;-
USR::
        MOV     2(R5),R2                ; Get FAB pointer
;+
;       Walk down through the linked list of XABs ( if any ) and
;       insert the PRO XAB at the end.
;-
        $FETCH      R3,XAB,R2           ; Get the first XAB addr if any
        BEQ     2$                      ; BR if none
1$:     $FETCH      R1,NXT,R3           ; Get the next XAB on the list
        BEQ     3$                      ; If none left BR
        MOV     R1,R3                   ; R3 = current XAB address
        BR      1$                      ; Cont until done
2$:     $STORE      #PROCOD,XAB,R2      ; Store our XAB address in the FAB
        BR      4$                      ; Cont
3$:     $STORE      #PROCOD,NXT,R3      ; Store our XAB address in the Last XAB on list
4$:     $CREATE     R2                  ; Create the file
        MOV     O$STS(R2),R0            ; Get error status
        BLE     5$                      ; Quit on error
        MOV     4(R5),R1                ; Get rab RAB pointer
        $CONNECT R1                     ; Connect the RAB-FAB
        MOV     O$STS(R1),R0            ; Get error status
5$:     RETURN                          ; RETURN
```

```
;+
;       Set the protection code for the XAB
;-
PROCOD:
        XAB$B        XB$PRO
.IF       DF         RSX
        X$PRO        60942                     ; (R,RWED,R,R)
.IFF
        X$PRO        40                        ; Set protection
.ENDC
        XAB$E
        .END
```

12.9.13 The WINDOWSIZE Clause

The WINDOWSIZE clause specifies the number of block retrieval pointers in memory for the file. The WINDOWSIZE clause is valid only on RSX systems. WINDOWSIZE is not a file attribute, and therefore can be changed any time you open a file.

Retrieval pointers are associated with the file header and point to contiguous blocks on disk. By keeping retrieval pointers in memory, you can reduce the I/O associated with locating a record because the operating system does not have to access the file header for pointers as frequently. The number of retrieval pointers in memory at any one time is determined by the system default or by the value you supply in the WINDOWSIZE clause. The usual default number of retrieval pointers is 7. You can specify up to 127 retrieval pointers.

On RSTS/E systems, the number of pointers in a window block is fixed at 7. Thus, you cannot use the WINDOWSIZE clause. Use the CLUSTERSIZE clause to increase the number of contiguous blocks mapped by one retrieval pointer.

13
Formatting Output

The PRINT USING statement controls the appearance and location of data on a line of output. With it, you can create formatted lists, tables, reports, and forms. This chapter describes how to format data with the PRINT USING statement.

13.1 Introduction

The ability to format data with the PRINT USING statement is useful because the way in which BASIC–PLUS–2 displays data is often limited. For example, a program may use floating-point numbers to represent dollars and cents. The PRINT statement displays floating-point numbers with up to six digits of accuracy, and places the decimal point anywhere in that 6-digit field. In contrast, PRINT USING lets you display floating-point numbers in the following ways:

- Rounded to two decimal places
- Vertically aligned on the decimal point
- Preceded by a dollar sign
- With commas every third digit to the left of the decimal point

Formatting monetary values in this way provides a much more readable report. Another use for formatted numeric values might be to print checks on a computer's line printer. PRINT USING lets you print numbers with a dollar sign and an asterisk-filled field preceding the first digit.

PRINT USING also formats string data. With it you can left- and right-justify string expressions, or center a string expression over a specified column position. Further, the PRINT USING statement can contain string literals. These are strings that do not control the format of a print item, but instead are printed exactly as they appear in the format string.

13.2 Using Format Strings

Format strings determine the way in which items are to be printed in the output file. Format strings can be the following:

- String variables
- String literals
- Named string constants
- A combination of the above

The PRINT USING statement must contain one or more format strings. Each format string is made up of one *format field*. Each format field controls the output of one print item and can contain only certain characters, as described throughout the chapter.

The PRINT USING statement must also contain a list of items you want printed. To format print items, you must separate them with commas or semicolons. Separators between print items do not affect output format as they do with the PRINT statement. However, if a comma or semicolon follows the last print item, BASIC–PLUS–2 does not return the cursor or print head to the beginning of the next line after it prints the last item in the list.

When BASIC–PLUS–2 encounters an invalid character within the current format field, it automatically ends the format field. Therefore, you do not need to delimit between format fields. The character that terminates the previous field can be either a new format field or a string literal.

In the following example, the first three characters in the format string (###) make up a valid numeric format field. The fourth character (A) is invalid in a numeric format field; therefore, BASIC–PLUS–2 ends the first format field after the third character. BASIC–PLUS–2 continues to scan the format string, searching for a character that begins a format field. The first such character is the number sign at character position seven. Therefore, the characters at positions four, five, and six are treated as a string literal. The characters at positions seven, eight, and nine make up a second valid numeric format field.

Example

```
10    PRINT USING "###ABC###", 123, 345
```

Output

```
123ABC345
```

When the statement executes, BASIC–PLUS–2 prints the first number in the list using the first format field, then prints the string literal ABC, and finally prints the second number in the list using the second format field. If you were to supply a third number in the list, BASIC–PLUS–2 would reuse the first format string. This is called *reversion.*

Example

```
10    PRINT USING "###ABC###", 123, 345, 564
```

Output

```
123ABC345
564ABC
```

Because any character not part of a format field is printed just as it appears in the format field, you can use a space or multiple spaces to separate format fields in the format string as shown in the following example.

Example

```
10    DECLARE STRING CONSTANT format_string = "###.##   ###.##"
      DECLARE SINGLE A,B
      A = 2.565
      B = 100.350
      PRINT USING format_string, A, B, A, B
```

Output

```
2.57   100.35
2.57   100.35
```

When the BASIC–PLUS–2 compiler encounters the PRINT USING statement, BASIC–PLUS–2 prints the value of *A* (rounded according to PRINT USING rules), three spaces, then the value of *B*. BASIC–PLUS–2 prints the three spaces because they are treated as a string literal in the format string. Notice that when BASIC–PLUS–2 reuses a format string, it begins on a new line.

13.3 Printing Numbers

With the PRINT USING statement, you can specify the following:

- The number of digits to print, thus rounding the number to a given place
- The decimal point location, thus vertically aligning numbers at the decimal point
- Special symbols, including trailing minus signs, asterisk-filled number fields, floating currency symbols, embedded commas, and E notation
- Debits and credits

- Leading zeros or leading spaces
- Blank-if-equal-to-zero fields
- A special character that is to be printed as a literal

Unlike the PRINT statement, PRINT USING does not automatically print a space before and after a number. Unless you reserve enough digit positions to contain the integer portion of the number (and a minus sign, if necessary), BASIC–PLUS–2 prints a percent sign (%) and displays the number in PRINT format.

13.3.1 Specifying the Number of Digits

You reserve places for digits by including a number sign (#) for each digit position. If you print negative numbers, you must also reserve a place for the minus sign.

Example

```
10     PRINT USING "###",123      !Three places reserved
20     PRINT USING "#####",12345  !Five places reserved
30     PRINT USING "####",-678    !Four places reserved
       END
```

Output

```
123
12345
-678
```

If there are not enough digits to fill the field, BASIC–PLUS–2 prints spaces before the first digit.

Example

```
10     format_string$ = "#####"
       PRINT USING format_string$, 1
       PRINT USING format_string$, 10
       PRINT USING format_string$, -1709
       PRINT USING format_string$, 12345
       END
```

Output

```
    1
   10
-1709
12345
```

If you have not reserved enough digits to print the fractional part of a number, BASIC–PLUS–2 rounds the number to fit the field.

Example

```
10    PRINT USING "###",126.7
      PRINT USING "#",5.9
      PRINT USING "#",5.4
      END
```

Output

```
127
6
5
```

If you have not reserved enough places to print a number's integer portion, BASIC–PLUS–2 prints a percent sign (%) as a warning symbol followed by the number in PRINT statement format. After BASIC–PLUS–2 prints the number, it completes the rest of the list in PRINT USING format.

In the following example, PRINT USING displays the first number. Because there are not enough places to the left of the decimal point to display a 3-digit number, BASIC–PLUS–2 prints the second number in PRINT statement format, with a space before and after, but includes a warning sign (%).

Example

```
10    PRINT USING "###", 256
      PRINT USING "##", 256
      END
```

Output

```
256
% 256
```

13.3.2 Specifying Decimal Point Location

The decimal point's position in the format string determines the number of reserved places on either side of it. If the print item's fractional part does not use all of the reserved places to the right of the decimal point, BASIC–PLUS–2 fills the remaining spaces with zeros.

Example

```
10    DECLARE STRING CONSTANT FM = "##.###"
20    PRINT USING FM, 15.72
30    PRINT USING FM, 39.3758
40    PRINT USING FM, 26
```

Output

```
15.720
39.376
26.000
```

If there are more fractional digits than reserved places to the right of the decimal point, BASIC–PLUS–2 rounds the number to fit the reserved places. Note that there must be enough places reserved to the left of the decimal point for the integer portion of the number. Otherwise, BASIC–PLUS–2 prints the number in PRINT format preceded by a warning sign (%). The following example shows how PRINT USING rounds numbers when you specify decimal point location.

Example

```
10      PRINT USING "##.##", 25.789
        PRINT USING "##.###", 100.2
        PRINT USING "#.##",.999
        END
```

Output

```
25.79
% 100.2
1.00
```

BASIC–PLUS–2 always fills all reserved spaces to the left of the decimal point with specified digits, spaces, or the minus sign.

Example

```
10      PRINT USING "##.##", 5.25
        PRINT USING "##.##", -5.25
        PRINT USING "###.##,-5.25
        END
```

Output

```
 5.25
-5.25
-5.25
```

13.3.3 Printing Numbers with Special Symbols

Special symbols let you print numbers with trailing minus signs, asterisk-fill fields, floating currency symbols, commas, or E notation. You can also specify debits, credits, leading zeros, leading blanks, and blank-if-zero fields. Table 13–1 summarizes these special characters.

Table 13–1 Format Characters for Numeric Fields

Character	Effect on Format
# number sign	Reserves a place for one digit.
. decimal point (period)	Determines decimal point location and reserves a place for the radix point.
, comma	Prints a comma before every third digit to the left of the decimal point and reserves a place for one digit or digit separator.
** two asterisks	Print leading asterisks before the first digit and reserve places for two digits.
$$ two dollar signs	Print a currency symbol before the first digit. They also reserve places for the currency symbol and one digit. By default, the currency symbol is a dollar sign. To change the currency symbol, see Section 13.3.3.3.
^^^^ four carets	Print a number in E (exponential) format and reserve four places for E notation.
– minus sign	Prints a trailing minus sign for negative numbers. Printing a negative number in an asterisk-fill or a currency field requires that the field also have a trailing minus sign or credit/debit character.
<0> Zero in angle brackets	Prints leading zeros instead of leading spaces.
<%> Percent sign in angle brackets	Prints all spaces in the field if the value of the print item, when rounded to fit the numeric field, is zero.
<CD> CD in angle brackets	Prints credit and debit characters immediately following the number. BASIC–PLUS–2 prints CR for negative numbers and zero, and DR for positive numbers.
_ Underscore	Specifies that the next character is a literal, not a formatting character.

13.3.3.1 Commas

You can place a comma anywhere in a number field to the left of the decimal point or to the right of the field's first character. A comma cannot start a format field. BASIC–PLUS–2 prints a comma to the left of every third digit from the decimal point. If there are fewer than four digits to the left of the decimal point, BASIC–PLUS–2 omits the comma.

Example

```
10      PRINT USING "##,###",10000
        PRINT USING "##,###",759
        PRINT USING "$$#,###.##",25694.3
        PRINT USING "**#,###",7259
        PRINT USING "####,#.##",25239
        END
```

Output

```
10,000
   759
$25,694.30
**7,259
25,239.00
```

13.3.3.2 Asterisk Fill Fields

To print asterisks (*) before the first digit of a number, you must start the field with two asterisks.

Example

```
10      DECLARE STRING CONSTANT FM = "**##.##"
20      PRINT USING FM, 1.2
30      PRINT USING FM, 27.95
40      PRINT USING FM, 107
50      PRINT USING FM, 1007.5
60      END
```

Output

```
***1.20
**27.95
*107.00
1007.50
```

Note that the asterisks reserve two places as well as cause asterisk fill.

To specify a negative number in an asterisk-fill field, you must place a trailing minus sign in the field. The trailing minus sign must be the last character in the format string.

Example

```
10      DECLARE STRING CONSTANT FM = "**##.##-"
        PRINT USING FM, 27.95
        PRINT USING FM, -107
        PRINT USING FM, -1007.5
        END
```

Output

```
**27.95
*107.00-
1007.50-
```

If you try to print a negative number in an asterisk fill field that does not include a trailing minus sign, BASIC–PLUS–2 signals "PRINT USING format error" (ERR=116).

You cannot specify both asterisk-fill and zero-fill for the same numeric field.

13.3.3.3 Currency Symbols

To print a currency symbol before the first digit of a number, you must start the field with two dollar signs. If the data contains both positive and negative numbers, you must include a trailing minus sign.

Example

```
10      DECLARE STRING CONSTANT FM = "$$##.##-"
        PRINT USING FM, 77.44
        PRINT USING FM, 304.55
        PRINT USING FM, 2211.42
        PRINT USING FM, -125.6
        PRINT USING FM, 127.82
        END
```

Output

```
 $77.44
$304.55
% 2211.42
$125.60-
$127.82
```

Note that the dollar signs reserve places for the currency symbol and only one digit; the dollar sign is always printed. (Hence the warning indicator (%) when the third PRINT USING statement executes.) Contrast this with the asterisk-fill field, where BASIC–PLUS–2 prints asterisks only when there are leading spaces.

If you try to print a negative number in a dollar sign field that does not include either a trailing minus sign or the CR/DR formatting character, BASIC–PLUS–2 signals "PRINT USING Format error" (ERR=116).

13.3.3.4 Negative Fields

To allow for a field containing negative values, you must place a trailing minus sign in the format field. A negative format field causes the value to be printed with a trailing minus sign. You can also denote negative fields with CR and DR. See Section 13.3.3.8 for more information.

You must use a trailing minus or the CR/DR formatting character to indicate a negative number in an asterisk fill or floating dollar sign field.

For fields with trailing minus signs, BASIC–PLUS–2 prints a minus sign after negative numbers as shown in Example 1, and a space after positive numbers as shown in Example 2:

Example 1

```
10     !Standard field
       PRINT USING "###.##",-10.54
       PRINT USING "###.##",10.54
       END
```

Output

```
-10.54
 10.54
```

Example 2

```
10     !Fields with Trailing Minus Signs
       PRINT USING "##.##-",-10.54
       PRINT USING "##.##-",10.54
       END
```

Output

```
10.54-
10.54
```

13.3.3.5 E (Exponential) Format

To print a number in E format, you must place four carets (^^^^) at the end of the field. The carets reserve space for the following:

- The capital letter E
- A plus or minus sign (which indicates a positive or negative exponent)
- An exponent (the exponent is 2 digits for single and double)

In exponential format, BASIC–PLUS–2 does not pad the digits to the left of the decimal point. Instead, the most significant digit shifts to the leftmost place of the format field, and the exponent compensates for this adjustment.

Example

```
10    PRINT USING "###.##^^^^",5
      PRINT USING "###.##^^^^",1000
      PRINT USING ".##^^^^",5
      END
```

Output

```
 500.00E-02
 100.00E+01
.50E+01
```

If you use fewer than four carets, the number does not print in E format; the carets print as literal characters. If you use more than four carets, BASIC–PLUS–2 prints the number in E format and includes the extra carets as a string literal.

Example

```
10    PRINT USING "###.##^^^",5
20    PRINT USING "###.##^^^^^",5
30    END
```

Output

```
  5.00^^^
500.00E-02^
```

You must reserve a place for a minus sign to the left of the decimal point to display negative numbers in exponential format. If you do not, BASIC–PLUS–2 prints a percent sign (%).

You cannot use exponential format with asterisk fill, floating dollar sign, or trailing minus formats.

13.3.3.6 Leading Zeros

To print leading zeros in a numeric field, you must start the format field with a zero enclosed in angle brackets (<0>). These characters also reserve one place for a digit.

Example

```
10    DECLARE STRING CONSTANT FM = "<0>####.##"
      PRINT USING FM, 1.23, 12.34, 123.45, 1234.56, 12345.67
```

Output

```
00001.23
00012.34
00123.45
01234.56
12345.67
```

When you specify zero-fill, you cannot specify asterisk-fill or floating-dollar sign format for the same field.

13.3.3.7 Blank-If-Zero Fields

To make BASIC–PLUS–2 print a blank field for values which round to zero, you must start the numeric field with a percent sign (%) enclosed in angle brackets (<%>).

In the following example, PRINT USING displays spaces in each reserved position for the second and third items in the list. The value of the second item is zero, while the value of the third item becomes zero when rounded to fit the numeric field.

Example

```
10    DECLARE STRING CONSTANT FM = "<%>####.##"
20    PRINT USING FM, 1000, 0, .001, -5000
```

Output

```
1000.00


-5000
```

13.3.3.8 Debits and Credits

You can have BASIC–PLUS–2 use credit and debit notation to differentiate positive and negative numbers. To do this, you place <CD> (Credit/Debit) at the end of the numeric format string. This causes BASIC–PLUS–2 to print CR (Credit Record) after negative numbers and zero, and DR (Debit Record) after positive numbers.

Example

```
10    DECLARE STRING CONSTANT FM = "$$####.##<cd>"
20    PRINT USING FM, -552.35, 200, -5
```

Output

```
$552.35CR
$200.00DR
  $5.00CR
```

You cannot use a trailing minus sign and Credit/Debit formatting in the same numeric field.

13.4 Printing Strings

You can format strings with the PRINT USING statement when you specify the following:

- The number of characters
- Left-justified format
- Right-justified format

- Centered format
- Extended field format

Table 13–2 summarizes the format characters and their function. Note only uppercase letter format characters are valid in BASIC–PLUS–2.

Table 13–2 Format Characters for String Fields

Character	Effect on Format
' single quotation mark	Starts the string field and reserves a place for one character.
L (uppercase)	Left-justifies the string and reserves a place for one character.
R (uppercase)	Right-justifies the string and reserves a place for one character.
C (uppercase)	Centers the string in the field and reserves a place for one character.
E (uppercase)	Left-justifies the string; expands the field, as necessary, to print the entire string; and reserves a place for one character.
\ \ two backslashes	Reserves *n*+2 character positions, where *n* is the number of spaces between the two backslashes. PRINT USING left-justifies the string in this field. This formatting character is included for compatibility with BASIC-PLUS. It is recommended that you do not use this type of field for new program development.
! exclamation point	Creates a one-character field. The exclamation point both starts and ends the field. This formatting character is included for compatibility with BASIC-PLUS. It is recommended that you do not use this type of field for new program development. Instead, use a single quotation mark to create a one-character field.

You must start string format fields with a single quotation mark (') that reserves a space in the print field, followed by one of the following:

- A contiguous series of uppercase Ls for left-justified output
- A contiguous series of uppercase Rs for right-justified output
- A contiguous series of uppercase Cs for centered output
- A contiguous series of uppercase Es for extended field output

BASIC–PLUS–2 ignores the overflow of strings larger than the string format field except for extended fields. For extended fields, BASIC–PLUS–2 extends the field to print the entire string. If a string to be printed is shorter than the format field, BASIC–PLUS–2 pads the string field with spaces. For more information on extended fields, see Section 13.4.4.

A string field containing only a single quotation mark is a one-character string field. BASIC–PLUS–2 prints the first character of the string expression corresponding to a one-character string field and ignores all following characters.

Example

```
10      PRINT USING "'","ABCDE"
        END
```

Output

```
A
```

13.4.1 Left-Justified Format

BASIC–PLUS–2 prints strings in a left-justified field starting with the left-most character. BASIC–PLUS–2 pads shorter strings with spaces and truncates longer strings on the right to fit the field.

A left-justified field contains a single quotation mark followed by a series of Ls.

Example

```
10      PRINT USING "'LLLLLL","ABCDE"
        PRINT USING "'LLLL","ABC"
        PRINT USING "'LLLLL","12345678"
        END
```

Output

```
ABCDE
ABC
123456
```

13.4.2 Right-Justified Format

BASIC–PLUS–2 prints strings in a right-justified field starting with the right-most character. BASIC–PLUS–2 pads the left side of shorter strings with spaces. If a string is longer than the field, BASIC–PLUS–2 left-justifies and truncates the right side of the string.

A right-justified field contains a single quotation mark (') followed by a series of Rs.

Example

```
10     DECLARE STRING CONSTANT right_justify = "'RRRRR"
       PRINT USING right_justify,"ABCD"
       PRINT USING right_justify,"A"
       PRINT USING right_justify,"STUVWXYZ"
       END
```

Output

```
  ABCD
     A
STUVWX
```

13.4.3 Centered Fields

BASIC–PLUS–2 prints strings in a centered field by aligning the center of the string with the center of the field. If BASIC–PLUS–2 cannot exactly center the string, as is the case for a 2-character string in a 5-character field, BASIC–PLUS–2 prints the string one character off center to the left.

A centered field contains a single quotation mark followed by a series of Cs.

Example

```
10     DECLARE STRING CONSTANT center = "'CCCC"
20     PRINT USING center, "A"
       PRINT USING center, "AB"
       PRINT USING center, "ABC"
       PRINT USING center, "ABCD"
       PRINT USING center, "ABCDE"
30     END
```

Output

```
  A
 AB
 ABC
ABCD
ABCDE
```

If there are more characters than places in the field, BASIC–PLUS–2 left-justifies and truncates the string on the right.

13.4.4 Extended Fields

An extended field contains a single quotation mark followed by one or more Es. The extended field is the only field that automatically prints the entire string. In addition, note the following:

- If the string is smaller than the format field, BASIC–PLUS–2 left-justifies the string as in a left-justified field.

- If the string is longer than the format field, BASIC–PLUS–2 extends the field and prints the entire string.

Example

```
10    PRINT USING "'E", "THE QUICK BROWN"
      PRINT USING "'EEEEEEE', "FOX"
      END
```

Output

```
THE QUICK BROWN
FOX
```

The following example uses left-justified, right-justified, centered, and extended fields.

Example

```
10    PRINT USING "'LLLLLLLLL","THIS TEXT"
      PRINT USING "'LLLLLLLLLLLLL","SHOULD PRINT"
      PRINT USING "'LLLLLLLLLLLLL",'AT LEFT MARGIN'
      PRINT USING "'RRRR","1,2,3,4"
      PRINT USING "'RRRR",'1,2,3'
      PRINT USING "'RRRR',"1,2"
      PRINT USING "'RRRR","1"
      PRINT USING "'CCCCCCCCC","A"
      PRINT USING "'CCCCCCCCC","ABC"
      PRINT USING "'CCCCCCCCC","ABCDE"
      PRINT USING "'CCCCCCCCC","ABCDEFG"
      PRINT USING "'CCCCCCCCC","ABCDEFGHI"
      PRINT USING "'LLLLLLLLLLLLLLLL',"YOU ONLY SEE PART OF THIS"
      PRINT USING "'E","YOU CAN SEE ALL OF THE LINE WHEN EXTENDED"
      END
```

Output

```
THIS TEXT
SHOULD PRINT
AT LEFT MARGIN
1,2,3
1,2,3
  1,2
    1
    A
   ABC
  ABCDE
 ABCDEFG
ABCDEFGHI
YOU ONLY SEE PART
YOU CAN SEE ALL OF THE LINE WHEN EXTENDED
```

13.5 Error Conditions

There are two types of PRINT USING error conditions: error and warning. BASIC–PLUS–2 signals an error if any of the following conditions exist:

- The format string is not a valid string expression
- There are no valid fields in the format string
- You specify a string for a numeric field
- You specify a number for a string field
- You separate the items to be printed with characters other than commas or semicolons
- A format field contains an invalid combination of characters
- You print a negative number in a floating dollar sign or asterisk-fill field without a trailing minus sign

BASIC–PLUS–2 issues a warning if a number does not fit in the field. If a number is larger than the field allows, BASIC–PLUS–2 prints a percent sign (%) followed by the number in the standard PRINT format and continues execution.

If a string is larger than any field other than an extended field, BASIC–PLUS–2 truncates the string and does not print the excess characters.

If a field contains an invalid combination of characters, BASIC–PLUS–2 does not recognize the first invalid character or any character to its right as part of the field. These characters may form another valid field or be considered text. If the invalid characters form a new valid field, a fatal error condition may arise if the item to be printed does not match the field.

The following examples demonstrate invalid character combinations in numeric fields.

Example 1

```
10    PRINT USING "$$**##.##",5.41,16.30
```

$$ forms a complete field and **##.## forms a second valid field. The first number (5.41) is formatted by the first valid field ($$). It prints as "$5". The second number (16.30) is formatted by the second field (**##.##) and prints as "**16.30".

Output 1

```
$5**16.30
```

Example 2

```
10     PRINT USING "##.#^^^",5.43E09
```

Because the field has only three carets instead of four, BASIC–PLUS–2 prints a percent sign and the number, followed by the ^^^.

Output 2

```
% .543E+10^^^
```

Example 3

```
10     PRINT USING "'LLEEE","VWXYZ"
```

You cannot combine two letters in one field. BASIC–PLUS–2 interprets EEE as a string literal.

Output 3

```
VWXEEE.
```

14
Compiler Directives

Compiler directives are instructions that tell BASIC–PLUS–2 to perform certain operations as it translates a source program. This chapter describes how to control program compilation using compiler directives.

14.1 Introduction

With compiler directives, you can do the following:

- Place program titles and subtitles in the header that appears on each page of the listing file
- Place a program version identification string in both the listing file and the object module
- Start or stop the inclusion of listing information for selected parts of a program
- Start or stop the inclusion of cross-reference information for selected parts of a program
- Include BASIC–PLUS–2 code from another source file or a text library
- Display a message at compile time
- Conditionally compile parts of a program
- Terminate compilation

All compiler directives:

- Must begin with a percent sign (%)
- Can be preceded by an optional line number
- Must be the only text on the line (except for %IF-%THEN-%ELSE-%END %IF)
- Cannot appear within a quoted string
- Cannot follow an END, END SUB, or END FUNCTION statement

14.2 Controlling the Compilation Listing

Listing directives let you control the content and appearance of the compilation listing. There are eight compiler listing directives:

- %TITLE (Places a title string on the first line of the listing header)
- %SBTTL (Places a subtitle string on the second line of the listing header)
- %IDENT (Places an identification string on the second line of the listing header and within the object module)
- %PAGE (Causes BASIC–PLUS–2 to skip to top-of-form in the output listing)
- %NOLIST (Causes BASIC–PLUS–2 to stop accumulating information for the output listing)
- %LIST (Causes BASIC–PLUS–2 to resume accumulating information for the output listing)
- %NOCROSS (Causes BASIC–PLUS–2 to stop accumulating cross-reference information for the output listing)
- %CROSS (Causes BASIC–PLUS–2 to resume accumulating cross-reference information for the output listing)

These directives are described in the following sections.

These listing control directives have no affect if no source program listing is being produced. Similarly, the %CROSS and %NOCROSS directives have no affect if no cross-reference listing is being produced. However, the %IDENT directive places the specified text in the object module whether or not a listing is produced. For more information on how these directives affect your source code, see the *BASIC–PLUS–2 Reference Manual.*

14.2.1 The %TITLE and %SBTTL Directives

The %TITLE directive lets you specify a line of text that appears on the first line of every page in the compilation listing. This text line is a quoted string of up to 48 characters and normally contains the source program title and other information.

If the %TITLE directive is the first source text in a module, then the quoted string appears in the first line of every page of the compilation listing. Otherwise, the quoted string appears in the first line of every subsequent page in the compilation listing. That is, if BASIC–PLUS–2 encounters a %TITLE directive after it has begun creating a page in the output listing, the

title information will not appear on that page. Rather, it appears on all of the following pages until it encounters another %TITLE directive.

The quoted string appears in the first line of the listing header. %TITLE must appear on its own line. For example:

```
10     %TITLE "File OPEN Subprogram -- Author Hugh Ristics"
       SUB FILSUB (STRING F_NAME)
```

The %SBTTL directive lets you specify a line of text that appears on the second line of every page in the compilation listing (beneath the title). If BASIC–PLUS–2 encounters a %SBTTL directive after it has begun creating a page in the output listing, the subtitle information will not appear on that page. Rather, it appears on all following pages until it encounters another %SBTTL or %TITLE directive. If you want the subtitle to appear on the first page, the %SBTTL directive must appear directly after the %TITLE directive.

Any number of %SBTTL directives can appear in a source file; thus, you can use subtitle text to identify parts of the source program. As in %TITLE, the text you use in %SBTTL must be a quoted string not exceeding 48 characters. The quoted string appears in the first line of the listing header. Note, however, that title and subtitle information only appears on listing pages that contain the actual source code.

The following example shows the use of both %TITLE and %SBTTL directives. The first line of the listing's first page contains "Payroll Program" and the second line contains "Constant Declarations." When BASIC–PLUS–2 encounters the %SBTTL directive, the second line on each subsequent page becomes "Subroutines." When BASIC–PLUS–2 encounters the %SBTTL directive, the second line on each subsequent page becomes "Error Handler."

Example

```
10     %TITLE "Payroll Program"
       %SBTTL "Constant Declarations"
   .
   .
   .
       %SBTTL "Subroutines"
   .
   .
   .
       %SBTTL "Error Handler"
   .
   .
   .
```

Note

You can use multiple %TITLE directives in a single source file; however, whenever BASIC–PLUS–2 encounters a %TITLE directive, the %SBTTL information is set to the null string. Therefore, if you want to display subtitle information, each new %TITLE directive should be accompanied by a new %SBTTL directive.

14.2.2 The %IDENT Directive

The %IDENT directive identifies the version of a program module. The identification text must be a quoted string of up to 6 characters. The information contained within the identification text appears in the listing file and the object module. Thus, the map file created by the task builder also contains this information.

The identification text appears in the first 6 character positions of the second line on each subsequent listing page. For instance, in the following example, the %IDENT information appears as the first entry on the second line of the listing. The information is also included in the object module if the compilation produces one. If the task builder generates a map listing, this information also appears there.

Example

```
10    %IDENT "V5.3"
      SUB PAY
   .
   .
   .
```

If your source module contains multiple %IDENT directives, BASIC–PLUS–2 signals a warning and uses the version specified in the first %IDENT directive.

14.2.3 The %PAGE Directive

The %PAGE directive causes BASIC–PLUS–2 to begin a new page in the listing file. In the following example, the %PAGE directives cause BASIC–PLUS–2 to skip to a new page in the listing file just before each new subtitle. Note that, in order to have title and subtitle information appear in the heading of each page, you cannot place a line number between the %PAGE, %TITLE, and %SBTTL directives.

Example

```
10    %TITLE "Payroll Program"
      %SBTTL "Constant Declarations"
    .
    .
    .
      %PAGE
      %SBTTL "Subroutines"
    .
    .
    .
      %PAGE
      %SBTTL "Error Handler"
    .
    .
    .
```

14.2.4 The %LIST and %NOLIST Directives

The %LIST directive causes BASIC–PLUS–2 to resume adding information to the listing file, while the %NOLIST directive causes BASIC–PLUS–2 to stop adding information to the listing file. %LIST and %NOLIST are complementary directives. Therefore, you can control which parts of the source program are to be listed.

In the following example, as soon as BASIC–PLUS–2 encounters the %LIST directive, it resumes adding new information to the listing file.

Example

```
10    %TITLE "Payroll Program"
      %SBTTL "Constant Declarations"
   .
   .
   .
      %NOLIST
   .
   .
   .
      %LIST
   .
   .
   .
      %PAGE
      %SBTTL "Subroutines"
   .
   .
   .
      %PAGE
      %SBTTL "Error Handler"
   .
   .
   .
```

If you have not requested the creation of a compilation listing, the %LIST and %NOLIST directives have no effect.

If a program line contains a syntax error, BASIC–PLUS–2 overrides the %NOLIST directive for that line and produces the normal error diagnostics in the listing file

14.2.5 The %CROSS and %NOCROSS Directives

The %CROSS directive causes BASIC–PLUS–2 to resume adding cross-reference information while the %NOCROSS directive causes BASIC–PLUS–2 to stop adding cross-reference information to the listing file. Therefore, you can specify that only certain parts of the source program are to be cross-referenced.

In the following example, as soon as BASIC–PLUS–2 encounters the %CROSS directive, it resumes adding new cross-reference information to the listing file.

Example

```
10    %TITLE "Payroll Program"
20    %SBTTL "Constant Declarations"
   .
   .
   .
30    %NOCROSS
   .
   .
   .
      %CROSS
   .
   .
   .
40    %PAGE
      %SBTTL "Subroutines"
   .
   .
   .
      %PAGE
      %SBTTL "Error Handler"
   .
   .
   .
```

If you have not requested the creation of a cross-reference listing, the %CROSS and %NOCROSS directives have no effect.

14.3 Accessing External Source Files (%INCLUDE)

The %INCLUDE directive lets you access BASIC–PLUS–2 source text from a file into the source program. The line on which a %INCLUDE directive resides can be continued, but cannot contain any other directives or statements.

If you are including a source text file, you must supply a file specification. If you do not provide a file type, BASIC–PLUS–2 uses the default type B2S. For example:

```
10    %INCLUDE  "SAMPLE.B2S"
```

The source files accessed with %INCLUDE cannot contain line numbers. This requirement means that all statements in the accessed file are associated with the BASIC–PLUS–2 line containing the %INCLUDE directive. A file accessed by %INCLUDE can itself contain a %INCLUDE directive.

When a program is compiled, BASIC–PLUS–2 inserts the included text at the point at which it encounters the %INCLUDE directive. The compilation listing identifies any text obtained from an included file by placing a mnemonic in the first character position of the line in which the text appears.

- *In* specifies text that was accessed from a source file.
- *I* tells you that the text was accessed with an %INCLUDE directive.
- *n* is a number that tells you the nesting level of the included text.

The %INCLUDE directive is useful when you want to share code among multiple program modules. To do this, you must first create a file which contains the shareable code, then include that file in all the modules that require it. Thus, you reduce the chance of a typographical error.

You can prevent the %INCLUDE file code from appearing in the compilation listing by preceding the %INCLUDE directive with a %NOLIST directive.

14.4 Controlling Compilation

BASIC–PLUS–2 lets you control the compilation of a program by creating and testing *lexical constants*. You create and assign values to lexical constants with the %LET directive. These constants are always WORD integers.

You control the compilation by using the %IF-%THEN-%ELSE-%END %IF directive to test these lexical constants. Thus, you can conditionally:

- Supply different values for program variables and constants
- Skip over part of a program
- Abort a compilation
- Include BASIC–PLUS–2 source code from another file
- Display informational messages during the compilation

BASIC–PLUS–2 also supplies the lexical built-in function %VARIANT that can be used to conditionally control compilation. For more information, see Section 14.4.2.

%IF-%THEN-%ELSE-%END %IF uses *lexical expressions* to determine whether to execute directives in the %THEN clause or the %ELSE clause. The following sections describe the use of:

- Lexical constants and expressions (%LET Directive)
- %VARIANT
- %ABORT
- %PRINT
- %IF-%THEN-%ELSE-%END %IF

14.4.1 Lexical Constants and Expressions (%LET)

The %LET directive creates and assigns values to lexical constants. Lexical constants are always WORD integers. These constants control the execution of the %IF-%THEN-%ELSE-%END %IF directive.

All lexical constants must be created with %LET before they can be used in a %IF-%THEN-%ELSE-%END %IF, and each lexical constant must be created with a separate %LET directive. All lexical constant names must also be preceded by a percent sign and cannot end with a dollar sign or percent sign.

A lexical expression can be any of the following:

- A lexical constant
- An integer literal
- A lexical built-in function (%VARIANT)
- Any combination of these, separated by logical, relational, or arithmetic operators

The %LET directive lets you create constants that control conditional compilation. For example:

```
10    %LET %debug_on = 0%
```

See Section 14.4.5 for an example of using %LET with %IF-%THEN-%ELSE.

14.4.2 The %VARIANT Directive

The %VARIANT directive is a built-in lexical function that returns an integer. The value of this returned integer is specified when you use the /VARIANT qualifier with the COMPILE, SET, or RUN commands in the BASIC–PLUS–2 environment. The default value for the %VARIANT function is zero. See Section Section 14.4.5 for an example of controlling compilations with the %VARIANT directive.

14.4.3 The %ABORT Directive

The %ABORT directive terminates the compilation and displays a message you provide.

The text must be a quoted string literal. This information is displayed on your terminal and in the compilation listing if one is requested. BASIC–PLUS–2 stops the compilation and terminates the listing file as soon as it encounters a %ABORT directive. This means that BASIC–PLUS–2 does not perform syntax checking on the remainder of the program. See Section 14.4.5 for an example of using %ABORT.

14.4.4 The %PRINT Directive

The %PRINT directive allows you to insert a message into your source code that the BASIC–PLUS–2 compiler displays at compilation time.

The text must be a quoted string literal. This information is displayed on your terminal and in the compilation listing if one is requested. BASIC–PLUS–2 prints the message specified as soon as it encounters a %PRINT directive. See Section 14.4.5 for an example of using %PRINT.

14.4.5 The %IF-%THEN-%ELSE-%END %IF Directive

The %IF-%THEN-%ELSE-%END %IF directive lets you conditionally do the following:

- Compile source text
- Execute another compiler directive

This directive differs from all others in that it can appear anywhere in a program where a space is allowed, except within a quoted string.

You must include %END %IF. Otherwise, the rest of the source program becomes part of the %THEN or %ELSE clause. You must also include a lexical expression and some BASIC–PLUS–2 source code.

The truth or falsity of the lexical expression determines whether BASIC–PLUS–2 compiles the source code in the %THEN clause or the %ELSE clause. If the lexical expression is true, BASIC–PLUS–2 neither compiles nor checks the syntax of source code in the %ELSE clause. If the lexical expression is false, BASIC–PLUS–2 neither compiles nor checks the syntax of source code in the %THEN clause.

The following example also uses the %VARIANT directive, which returns the value set by the SET VARIANT command:

Example

```
10    %IF (%VARIANT = 2%)
          %THEN DECLARE LONG int_array(100)
          %ELSE DECLARE WORD int_array(100)
      %END %IF
```

Because %IF can appear within a program line, you can express the same directive this way:

```
10    DECLARE %IF (%VARIANT=2%) %THEN LONG %ELSE WORD %END %IF int_array(100)
```

A %THEN or %ELSE clause can also contain other compiler directives. For example, the following program creates the lexical constant *%my_constant* and assigns it a value of eight. The %IF directive evaluates the conditional expression *((%my_constant + %VARIANT) >= 10%)*. If this expression is true, BASIC–PLUS–2 executes the %THEN clause, aborting the compilation and issuing an error message. If the expression is false, the next conditional expression *((%my_constant + %VARIANT) < 10%)* is evaluated. If this expression is true, BASIC–PLUS–2 then executes the %THEN clause, printing the specified message. If the expression is false, BASIC–PLUS–2 continues to compile your program without aborting the compilation.

Example

```
10    %LET %my_constant = 8%
      %IF ( (%my_constant + %VARIANT) >= 10% )%THEN
             %ABORT "Cannot compile with VARIANT >= 2"
             %ELSE %PRINT "Successful Compilation"
      %END %IF
```

The compilation listing shows you which clause was actually compiled.

15 Error Handling

The process of detecting and correcting errors that occur during program execution is called *error handling*. This chapter describes BASIC–PLUS–2 default error handling and how to handle run-time errors with your own *error handlers*.

15.1 Error Handlers

An error handler is a block of code that receives program control in the event of an error or unexpected event. If you do not supply a user-written error handler, the BASIC–PLUS–2 error handler usually receives program control when an error occurs (there are some types of errors that are handled only by the operating system).

The severity of an error determines whether the program aborts when the error occurs. There are four severity levels of errors:

- Information
- Warning
- Error (also called trappable)
- Fatal

Warning and information errors allow the program to continue executing; fatal errors always terminate program execution. Trappable errors can have either result, depending on the error handler in effect. If the default BASIC–PLUS–2 error handler is in effect, trappable errors cause the program to abort.

In a BASIC–PLUS–2 program, there are many possible levels of error handling. The following software components and program modules can each have an error handler:

- Mainline code
- User-written subprograms or external functions
- User-defined functions (DEF functions)

- The BASIC–PLUS–2 system

You can supply your own error handler for each of the first three components. The BASIC–PLUS–2 error handler is the default if you do not supply one.

When an error occurs in your program, BASIC–PLUS–2 stops executing the program and transfers control to the BASIC–PLUS–2 error handler. If your program includes a user-written error handler, and the error is trappable, the BASIC–PLUS–2 error handler then transfers control to the user error handler. However, if your program does not include a user error handler or the error is a nontrappable or fatal error, the BASIC–PLUS–2 error handler retains control.

The following sections describe both default and user-written error handlers.

15.1.1 BASIC–PLUS–2 Default Error Handling

BASIC–PLUS–2 provides default run-time error handling for all programs. If you do not provide your own error handlers, the default error handling procedures remain in effect throughout program execution.

When an error occurs in your program and the program does not contain a user-written error handler, BASIC–PLUS–2 diagnoses the error and displays a message telling you the nature and severity of the error, and the program line and module that caused it. The severity of an error determines whether or not the program aborts. When default error handling is in effect, fatal errors always terminate program execution, but program execution continues when warning and informational errors occur.

15.1.2 User-Written Error Handlers

It is good programming practice to anticipate certain errors and provide your own error handlers for them. User-written error handlers allow you to handle errors for a specified block of program statements as well as complete program units. Any program module can contain one or more error handlers. These error handlers test the error condition and include statements to be executed if an error occurs.

A user-written error handler can be used to do the following:

- Identify the error
- Indicate which program unit or statement caused the error
- Take appropriate action based on the nature of the error
- Clear the error condition
- Continue program execution

In BASIC–PLUS–2, only one error can be handled at a time. If an error is pending and a second error occurs, program execution always terminates immediately. Therefore, one of the most important functions of a user-written error handler is to clear the error condition so that subsequent errors can also be handled.

To transfer control to a user-written error handler, you specify a line number or label with the ON ERROR GOTO statement. When an error occurs, the block of code at the line number or label is executed.

The error handler can use conditional expressions to test an error and branch accordingly. In the following example, the error handler tests for two types of errors: an error that occurs when a record is longer than the length of the buffer, and an error generated by the error "End of file" (ERR=11).

Example

```
50      ON ERROR GOTO 19000
    .
    .
    .
19000   SELECT ERR
        CASE 161
                PRINT "Record too long"
                RESUME 650
        CASE 11
                PRINT "End of file"
                RESUME 32000
        CASE ELSE
                ON ERROR GOTO 0
        END SELECT
32000   CLOSE #1%
32767   END
```

Customarily, the ON ERROR GOTO statement is positioned before any other executable statement, and an error handler usually starts at line 19000.

The ON ERROR GOTO statement remains in effect after your program successfully handles an error. Therefore, if another error occurs in your program, control once again transfers to the specified line. If an error occurs within an error handler itself, control passes to the BASIC–PLUS–2 default error handler and program execution ends, usually with the initial error only partially processed. To avoid the possibility of your error handler causing an error, your error handler should be as simple as possible.

Two other common errors you can trap with error handlers are "Division by 0" and "Data format error." For instance, if your program is reading data from a file when one of these errors occurs, you can have it print an error message and skip to the next item; if the program is reading data that the user enters, you

can display a “Try again” message and reexecute the program lines requesting input.

Normally, you cannot trap fatal errors in an error handler, nor can you trap errors occurring in non-BASIC modules. However, in BASIC–PLUS–2 you can trap certain errors (for example, Ctrl/C interrupts) occurring in a MACRO–11 subprogram. See Chapter 11 for more information about MACRO–11 subprograms.

15.2 Identifying Errors

BASIC–PLUS–2 provides several built-in functions that return information about errors. You can use these functions inside your error handlers to get information about an error and then handle the error based on that information. These functions are as follows:

- ERR
- ERL
- ERN$
- ERT$

The following sections describe these functions.

15.2.1 Determining the Error Number (ERR)

The ERR function returns the number of the last error that occurred. If an error does not occur, ERR is undefined.

The error handler in the following example transfers control to line 420 when the program generates the error “Record already exists.” Line 19000 prints the value returned by the ERR function each time BASIC–PLUS–2 traps an error.

Example

```
19000   PRINT "ERROR NUMBER ";ERR
19010   SELECT ERR
        !Record already exists
        CASE 153
                PRINT "Choose new record"
                RESUME 420
        CASE ELSE
                ON ERROR GO BACK
        END SELECT
```

ERR remains defined as the number of the last error after control leaves the error handler. However, it is poor programming practice to refer to this variable outside the scope of an error handler because it can be changed at any time by an asynchronous error.

See B for a list of run-time errors and their numbers.

15.2.2 Determining the Error Line Number (ERL)

After your program generates an error, the ERL function returns the line number of the signaled error. The results of ERL are undefined if an error does not occur or if an error occurs in a subprogram not written in BASIC–PLUS–2. The ERL function, like the ERR function, allows you set up branching to one of several paths in the code.

The following error handler continues execution at different points in the program, depending on the value of ERL.

Example

```
80      ON ERROR GOTO 19000
        DECLARE INTEGER CONSTANT TRUE = -1%
   .
   .
   .
19000   SELECT TRUE
        CASE (ERR = 11) AND (ERL = 790)
        !Is error end-of file at line 790?
                PRINT "Completed"
                RESUME 32000
        CASE (ERR = 149) AND (ERL = 80)
        !Is error not-at-end-of-file on line 80?
                PRINT "CHECK ACCESS MODE"
                RESUME 32000
        CASE ELSE
        !Let BASIC--PLUS--2 handle any other errors
                ON ERROR GOTO 0
        END SELECT
32000   CLOSE #5
32767   END
```

In this example, if the error is “End of file” (ERR=11), the handler closes the file and exits. If the error is “Not at end of file” (ERR=149), the handler returns control to line 80. If any other error occurs, the error handler passes control to the BASIC–PLUS–2 default error handler.

ERL remains defined as the line number of the last occurring error after control leaves the error handler. However, it is poor programming practice to refer to this variable outside the scope of an error handler.

Note that if your program references the ERL function and you compile it with the /NOLINE qualifier, BASIC–PLUS–2 signals the message “ERL overrides /NOLINE” and the program is compiled with the /LINE qualifier.

If an error occurs in a subprogram, the value returned by ERL is the subprogram line number where the error was detected.

If you specify an ON ERROR GO BACK statement in a subprogram, control is transferred to the error handler of the calling program when an error occurs. For execution to continue, the RESUME statement in the calling program must reference the line number of the CALL statement that calls the subprogram. This is because you cannot specify a line number outside the current program module with a RESUME statement.

15.2.3 Determining Where the Error Occurred (ERN$)

The ERN$ function returns the name of the main program, SUB or FUNCTION subprogram or DEF function in which the error was detected. The results of ERN$ are undefined until the program generates an error.

In the following example, control passes to the main program for error handling if the error occurs in the module SUBARC. If the error is "Record already exists," execution resumes at line 32000. If any other error occurs, control passes to BASIC–PLUS–2 error handler.

Example

```
19000        IF ERN$ = "SUBARC"
                THEN PRINT "Error is ";ERR
             END IF

             PRINT "Returning to main program for error handling"
             ON ERROR GO BACK

19010        PRINT "Program module generating error is ";ERN$

19020        IF ERR = 153%
                THEN RESUME 32000
                    ELSE ON ERROR GO TO 0
             END IF

32000        CLOSE #2%
32767        END
```

15.2.4 Determining the Error Message Text (ERT$)

The ERT$ function returns the message text associated with a specified error number. The ERT$ function is not limited to the scope of the error handler; you can access the ERT$ function at any time.

The following example tests whether an error occurred in the DEF module TSLFE and, if it has, prints the text of the signaled error and resumes execution.

Example

```
19000       IF ERN$ = "TSLFE"
               THEN PRINT ERT$(ERR)
            END IF
            RESUME
```

Note that if an error occurs in a DEF declaration, RESUME without a line number causes execution to resume at the statement that invoked the function.

15.2.5 Ctrl/C Trapping

Error handling procedures are commonly used to trap user Ctrl/C responses. With Ctrl/C trapping enabled, control is transferred to an error handler if a user presses Ctrl/C during program execution. You enable Ctrl/C trapping in your program by invoking the built-in CTRLC function. For example:

```
Y% = CTRLC
```

Once the Ctrl/C is trapped, you can include routines to interact with the program, as shown in the following example:

Example

```
50       ON ERROR GOTO 19000
100      Y% = CTRLC
   .
   .
   .
460      OPEN 'FIL.DAT' FOR INPUT AS FILE #1%
470      INPUT "How many records"; REC.READ%
480      FOR I% = 1% TO REC.READ%
490             GET #1%
500             PRINT NA.ME$, ADDRE.SS$, EMP.CODE%
                PRINT
510      NEXT I%
   .
   .
   .
19000    IF (ERR = 28%)
           THEN Y% = CTRLC        !Re-enable Ctrl/C trapping
                PRINT "Current record is "; I%
            ELSE ON ERROR GOTO 0
         END IF
```

```
19010    INPUT "Do you wish to end processing?"; ANSWER$
19020    IF ANSWER$ = "Yes"
            THEN RESUME 32000
               ELSE RESUME
         END IF
   .
   .
   .
32000    CLOSE #1%
32766    PRINT "End of processing"
32767    END
```

Output

```
SMITH, DEXTER 231 COLUMBUS ST             09341

TRAVIS, JOHN  PO BOX 80                   64119

^C

The current record is 3

Do you wish to end processing? Yes
```

An error condition is pending until the error handler executes a RESUME statement. Therefore, if a second Ctrl/C is entered while the error handler is executing, control returns to the BASIC–PLUS–2 error handler, which terminates the program.

With Ctrl/C trapping enabled, a RESUME statement with no line number returns control to the line where the error occurred. Your program can then re-execute statements interrupted by the Ctrl/C.

To disable Ctrl/C trapping, use the RCTRLC function. For more information on the CTRLC and RCTRLC functions, see the *BASIC–PLUS–2 Reference Manual*.

15.3 Handling Errors in Multiple-Unit Programs

When an error occurs in a subprogram or function definition, control always passes to the error handler contained within that subprogram or function definition. If a subprogram or function definition does not contain an error handler, all error handlers for any outer program blocks are processed before the program reverts to BASIC–PLUS–2 default error handling.

The following rules apply to error handling in function definitions:

- To trap an error while a DEF function is active, include an error handler inside the DEF function. When you include an error handler inside a DEF function, the associated handler remains in effect until your program leaves the DEF function.

- An error handler in a DEF function does not permanently override an error handler in the main program. BASIC–PLUS–2 saves the error handler in the main program when you invoke a DEF function, and restores it when you return.

If an ON ERROR GO BACK statement is specified in a subprogram or function definition, it transfers control to the error handler of the calling program. If ON ERROR GO BACK is specified in a main program module, it transfers control to the BASIC–PLUS–2 default error handler.

Example

```
100       ON ERROR GOTO 19000
   .
   .
   .
1000      A% = Fnin_put%("Prompt")
   .
   .
   .
15000     DEF Fnin_put%(P%)
          ON ERROR GOTO 15090
15010     PRINT P$
          INPUT LINE_IN$
15020     Fnin_put% = Val%(Line_in$)
          FNEXIT
15090     IF ERL = 15010
             THEN
                PRINT "Retry"
                RESUME 15010
             ELSE
                ON ERROR GO BACK
          END IF
   .
   .
   .
16000     END DEF
19000     PRINT "Error"; ERT$(ERR);
```

```
19010       IF ERN$ = "Fnin_put"
               THEN
                  PRINT "In function"
                  RESUME 1200
               ELSE
                  PRINT "In main"
                  RESUME 1000
            END IF
    .
    .
    .
32767       END
```

15.4 Returning to BASIC–PLUS–2 Error Handling

The ON ERROR GOTO 0 statement disables program error trapping and returns control to BASIC–PLUS–2 error handling. The BASIC–PLUS–2 error handler displays error messages, stops program execution, and prints a fatal error message.

If an error is pending, execution of the ON ERROR GOTO 0 statement returns control to BASIC–PLUS–2 error handling immediately. If no error is pending, an ON ERROR GOTO 0 statement disables your error handler. The BASIC–PLUS–2 error handler handles all subsequent errors until another ON ERROR statement is executed.

In the following example, when an error occurs, control is transferred to line 19000. If the error is "End-of-file" (ERR=11), the error handler passes control to line 32000, which closes the file. If any other error occurs, the ELSE clause transfers control to the BASIC–PLUS–2 error handler, which prints information about the error.

Example

```
10          ON ERROR GOTO 19000
20          OPEN 'FILE.LIS' FOR INPUT AS FILE #2%
30          LINPUT #2%, A$
40          PRINT A$
50          GOTO 30
19000       IF (ERR = 11%) AND (ERL = 30%)
               THEN
                  RESUME 32000
               ELSE
                  ON ERROR GOTO 0
            END IF
32000       CLOSE #2%
32767       END
```

You can use the ON ERROR GOTO and ON ERROR GOTO 0 statements throughout your program to turn an error handler on and off. In this way, your program can handle certain errors, and BASIC–PLUS–2 can handle the rest.

15.5 Leaving an Error Handler

The RESUME statement clears the error condition and passes control to a line number or to the program block in which the error occurred. An error handler must end with either a RESUME statement, an ON ERROR GOTO 0 statement, or an ON ERROR GO BACK statement. If it does not, the BASIC–PLUS–2 error handler aborts your program with the fatal error "Error trap needs RESUME" once BASIC–PLUS–2 encounters an END, END SUB, END DEF, END PROGRAM or END FUNCTION statement. You can resume to any line number in the same module as the RESUME statement unless that line number is inside a DEF function.

The RESUME statement in the following error handler resumes program execution at line 32767 when one of the specified errors occur.

Example

```
19000    IF (ERR = 11%) AND (ERL = 30%)
             THEN
                CLOSE #2%
                RESUME 32767
             ELSE
                ON ERROR GOTO 0
         END IF
32000    CLOSE #2%
32767    END
```

If you do not specify a line number with the RESUME statement, control is passed to the beginning of the program block where the error occurred. A program block can begin with either a line number or a label.

- If you resume execution at a multi-statement line, execution begins at the first statement after the line number or label—not necessarily at the statement that generated the error.
- If an entire FOR, WHILE, UNTIL, or SELECT loop block is associated with a single line number or label and an error occurs within that loop block, RESUME without a line number transfers control to the statement immediately following the FOR, WHILE, UNTIL or SELECT statement, not to the line number or label.

If general, if you specify a RESUME statement without a line number, be sure to supply a separate line number for every statement that may generate an error.

When resuming to a FOR, WHILE, UNTIL, or SELECT loop, the starting, ending, and STEP values of the loop are not reinitialized.

Even though the line at which you want to resume program execution may directly follow an error handler, you cannot "fall through" an error handler to a END SUB, END DEF, END PROGRAM, END FUNCTION, or END statement. You must either resume to a line number or return to system error handling.

Use a RESUME, ON ERROR GO BACK, or ON ERROR GOTO 0 statement instead of a GOTO statement to exit from an error handler. If you specify a GOTO statement inside an error handler, it will leave the error condition pending and the next error will abort the program.

An error handler should correct the condition that caused the error. If it does not, the program will continue to generate the error without handling it successfully.

In the following example, when a record name is not found, the error handler corrects the error by prompting for a new record name.

Example

```
5        ON ERROR GOTO 19000
   .
   .
   .
330      INPUT "What record do you want"; Target.name$
340      FIND #12%, KEY #3% EQ Target.name$
   .
   .
   .
19000    IF (ERR = 153%)
            THEN PRINT "Name invalid"
         END IF
19010    RESUME 330
   .
   .
   .
32767    END
```

For more information on the RESUME statement, see the *BASIC–PLUS–2 Reference Manual*.

16
Instruction and Data Space

This chapter describes how to use Instruction and Data Space (I- and D-Space) to improve system performance while running large BASIC–PLUS–2 tasks.

16.1 Introduction

Some PDP–11 computers include one set of active page registers (APRs) for instructions and one set for data. Normally, only one set of registers is used for both instructions and data; this is called *overmapping*. Instruction and Data Space (I- and D-Space) allows you to use both the instruction and the data registers for task execution. As a result, 32K words are made available for instruction addresses and another 32K words are made available for data addresses. This increased memory allows you to execute large tasks more efficiently. Note that you cannot use the BASIC–PLUS–2 resident library to link programs that use I- and D-Space support.

16.2 Building Tasks in Instruction and Data Space

You can build tasks in I- and D-Space by adding the /IDS qualifier to the BUILD or SET commands in the BASIC environment. For example:

```
BUILD /IDS MYPROG
```

The /IDS qualifier remains in effect until you override it with another build operation. When you specify the /IDS qualifier with the BUILD command, it removes all references to the BASIC–PLUS–2 memory-resident library from the Task Builder Command (CMD) file. It is recommended that you do not use the BASIC–PLUS–2 memory-resident library for tasks built in I- and D-space.

Note

You must recompile programs that were compiled prior to Version 2.3 of BASIC–PLUS–2 if you want to task build in I- and D-Space.

When you build a task with I- and D-Space support, you are building a task with two address windows: one in I-Space and one in D-Space. You can see the effects of I- and D-Space by adding the /MAP qualifier to the BUILD command. For example, the following is a map file generated by the following BUILD command:

```
BUILD /IDS /MAP MAINPROG
```

Example 16–1 Map File Illustrating Instruction and Data Space

```
MAINPROG.TSK   Memory allocation map  TKB M43.00      Page 1
                     20-APR-91   09:03

Partition name : GEN
Identification : 000220
Task  PPN      : [254,240]
Stack    limits: 001000 001777 001000 00512.
PRG xfr address: 025600
Task attributes: ID
Total address windows: 2.
Task   extension   : 512. words
Task  image  size  : 5728. words, I-Space
                     1376. words, D-Space
Total  task  size  : 5728. words, I-Space
                     1888. words, D-Space
Task Address limits: 000000 026277 I-Space
                     000000 005253 D-Space
R-W disk blk limits: 000002 000036 000035 00029.

*** Root segment: MAINPROG


R/W mem  limits: 000000 027275 027276 11966. I-Space
                 000000 005251 005252 02730. D-Space

Disk blk limits: 000002 000030 000027 00023. I-Space
                 000031 000036 000006 00006. D-Space
    .
    .
    .
```

In this example, the task size is reported as 5728 words for I-Space and 1888 words for D-Space. Assuming that all 32,768 words are available for the task, the task can therefore be extended 27,040 words in I-Space and 30,880 words in D-Space.

The following sections describe some areas where you want to exercise care when using I- and D-Space.

16.2.1 MACRO Subprograms

If you program with MACRO subprograms and you want to use I- and D-Space support, your MACRO subprograms should reflect the I- and D-Space attribute of the PSECTs. This means that you may have to rewrite the MACRO routines.

The following is an example of a MACRO subprogram that is not built with I- and D-Space:

```
.PSECT CODE1
    .
    .
    .
 MOV   #3, DEST
    .
    .
    .
 DEST: .WORD 0
    .
    .
    .
```

The following is the same MACRO subprogram, rewritten with I-PSECTs and D-PSECTs:

```
.PSECT CODE1,I
    .
    .
    .
 MOV  #3,DEST
    .
    .
    .
.PSECT CODE2,D
    .
    .
    .
 DEST: .WORD 0
    .
    .
    .
```

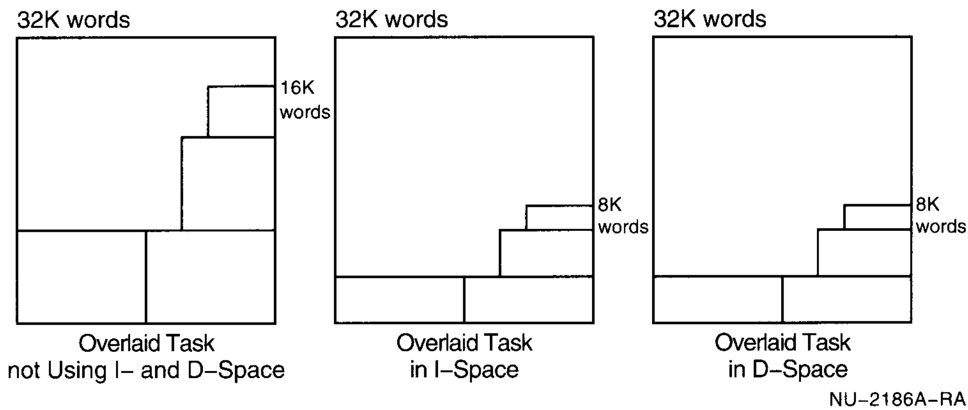
16.2.2 Overlaid Tasks

If you are building a complex overlaid task, you may want to refrain from using I- and D-Space, as problems can result. One such problem is that even though you save virtual memory when you use I- and D-Space, you double the number of program segments in your program. Therefore, if you had *n* number of segments in your program before invoking I- and D-Space, you have *n* number of segments in I-Space, plus *n* number of segments in D-Space after invoking I- and D-Space.

This increase in the number of program segments can causes the Task Builder to signal the error "No virtual memory storage available" when the task is task-built. You can avoid this problem by using the slow Task Builder. See the *RSTS/E Task Builder Reference Manual* or the *RSX-11M/M-PLUS Task Builder Reference Manual* for more information.

Figure 16–1 illustrates how tasks are overlaid in memory with and without I- and D-Space enabled.

Figure 16–1 Task Layout



Also, note that because the size of the task is extended when you use I- and D-Space, you may have to re-overlay your program in order to simplify your ODL file.

17
Advanced Input-Output

This chapter describes some of the more advanced I/O features available in BASIC–PLUS–2. For more information on I/O to RMS disk files, see Chapter 12.

17.1 Introduction

This chapter discusses the following topics:

- RMS I/O to ANSI magnetic tapes
- Device-specific I/O to magnetic tapes (including TK50 devices)
- Devce-specific I/O to disks and unit record devices
- Network I/O

When you do not specify a file name in the OPEN statement, the I/O you perform is said to be *device-specific*. This means that read and write operations (GET and PUT statements) are performed directly to or from the device. For example:

```
OPEN "MM0:" FOR OUTPUT AS FILE #1
OPEN "MM1:PARTS.DAT" FOR INPUT AS FILE #2, SEQUENTIAL
```

Because the file specification in the first line does not contain a file name, the OPEN statement opens the tape drive for device-specific I/O. The second line opens an ANSI-format tape file using RMS because a file name is part of the file specification.

The following sections describe both I/O to ANSI-format magnetic tapes and device-specific I/O to magnetic tape, unit record, and disks devices.

17.2 RMS I/O to Magnetic Tape

BASIC–PLUS–2 supports I/O to ANSI-formatted magnetic tapes. When performing I/O to ANSI-formatted magnetic tapes, you can read or write only one file to a magnetic tape at a time, and the files are not available to other users. ANSI tape files are RMS sequential files.

17.2.1 Allocating a Tape

You should allocate the tape unit to your process before starting file operations. You can allocate a device only when it is not allocated by someone else. The DCL command ALLOCATE reserves a device for your use. The following is an example of the ALLOCATE command. This command assigns tape drive MM0: to your process.

```
$ ALLOCATE MM0:
```

Once you allocate the tape unit, you can physically mount the tape on the tape drive. See the operation manual for your tape drive for instructions on how to physically mount the tape.

See the *RSX–11M–PLUS Command Language Manual* or the *RSTS/E System User's Guide* for more information on the ALLOCATE command.

17.2.2 Initializing a Tape on RSX–11M Systems

If you want to use RMS–11 and you are using a tape for the first time or you want to reuse a scratch tape, you must initialize it. Initializing a tape writes an ANSI-standard volume label and a dummy file that deletes all existing files on the tape. Therefore, you want to initialize only new tapes or scratch tapes and not tapes containing data you want to use.

In multiuser protection systems, you can initialize a tape only on tape drives allocated to you. You must have privileges to initialize a tape on RSX–11M systems that do not have multiuser protection.

To initialize a tape on RSX–11M systems, you specify the DCL command INITIALIZE. The following is an example of the INITIALIZE command:

```
$ INITIALIZE MYTAPE: PRG
```

This command initializes the tape assigned to the logical name MYTAPE: and specifies PRG as the volume label.

See the *RSX–11M–PLUS Command Language Manual* for more information on the INITIALIZE command.

17.2.3 Initializing a Tape on RSX–11M–PLUS Systems

If you want to use RMS–11 and you are using a tape for the first time or you want to reuse a scratch tape, you must initialize it. Initializing a tape writes an ANSI-standard volume label and a dummy file that deletes all existing files on the tape. Therefore, you want to initialize only new tapes or scratch tapes and not tapes containing data you want to use.

To initialize a tape on RSX–11M–PLUS systems, you specify the following sequence of DCL commands:

- The MOUNT command with the /FOREIGN qualifier
- The INITIALIZE command
- The DISMOUNT command
- The MOUNT command

These commands are described in the following sections.

17.2.3.1 The MOUNT Command

The DCL command MOUNT lets the system know that the tape is online and available for use. The MOUNT command also does the following:

- Verifies that the tape drive is not allocated to another user
- Allocates the tape drive to you if the tape drive is available
- Verifies that the tape is placed and loaded on the device you specify
- Checks that the tape has been initialized, if you use a tape that has already been initialized
- Specifies the tape's density and format
- Verifies that the volume label on the tape matches the volume label you specify

To initialize a tape on RSX–11M–PLUS systems, you specify the /FOREIGN qualifier to the MOUNT command. The /FOREIGN qualifier tells the system that the tape being mounted is not in files-11 format.

See the *RSX–11M–PLUS Command Language Manual* for more information on the MOUNT command.

17.2.3.2 The INITIALIZE Command

Once you specify the MOUNT command, you then initialize the tape with the DCL command INITIALIZE. If you specified a volume label with the MOUNT command, specify the same volume label with the INITIALIZE command. Nonprivileged users must specify a volume label. Note that this label is not the same as an ANSI-format volume label.

See the *RSX–11M–PLUS Command Language Manual* for more information on the INITIALIZE command.

17.2.3.3 The DISMOUNT Command

If you are initializing a tape, or after you finish performing file handling operations, you dismount the tape using the DCL command DISMOUNT. The DISMOUNT command does the following:

- Closes all open files
- Tells the system that the tape drive is logically offline
- Breaks the connection between the system's file system and the device

Dismounting a tape prevents a program from opening new files on that tape.

See the *RSX–11M–PLUS Command Language Manual* for more information on the DISMOUNT command.

17.2.3.4 Example of Initializing a Tape on RSX–11M–PLUS

The series of commands in the following example initialize the tape MYTAPE: on RSX–11M–PLUS:

```
$ ALLOCATE MYTAPE:
$ MOUNT /FOREIGN MYTAPE:
$ INITIALIZE MYTAPE: PRG
$ DISMOUNT MYTAPE: PRG
$ MOUNT MYTAPE:
```

In this example:

1. The ALLOCATE command allocates the tape drive assigned to logical name MYTAPE:.
2. The first MOUNT command specifies a foreign file structure.
3. The INITIALIZE command initializes the tape, writes an ANSI-format volume label, and uses PRG as the volume label to identify the tape.
4. The DISMOUNT command dismounts the tape.
5. The second MOUNT command allows the system to read the file structure and verifies the volume label.

To use a tape that has already been initialized, you need only issue the ALLOCATE and MOUNT commands. For example:

```
$ ALLOCATE MYTAPE:
$ MOUNT MYTAPE: PRG
```

In this example, the ALLOCATE command allocates the tape drive assigned to logical name MYTAPE: and tells the system that the tape on that tape drive is online and available for use, and verifies the volume label.

17.2.4 Initializing a Tape on RSTS/E Systems

If you want to use RMS–11 and you are using a tape for the first time or you want to reuse a scratch tape, you must initialize it. Initializing a tape writes an ANSI-standard volume label and a dummy file that deletes all existing files on the tape. Therefore, you want to initialize only new tapes or scratch tapes and not tapes containing data you want to use.

To initialize a tape on RSTS/E systems, you specify the following sequence of DCL commands:

- The INITIALIZE command with the /FORMAT=ANSI qualifier
- The MOUNT command

These commands are described in the following sections.

17.2.4.1 The INITIALIZE Command

The DCL command INITIALIZE with the /FORMAT=ANSI qualifier initializes a tape for use with RMS–11 files. Any tape that is not initialized with the INITIALIZE command is a foreign tape and requires you to add the /FORMAT=FOREIGN qualifier to the MOUNT command. In multiuser protection systems, you can initialize a tape only on tape drives allocated to you. You must have privileges to initialize a tape on RSTS/E systems that do not have multiuser protection.

See the *RSTS/E System User's Guide* for more information on the INITIALIZE command.

17.2.4.2 The MOUNT Command

The DCL command MOUNT lets the system know that the tape is online and available for use. The MOUNT command also does the following:

- Verifies that the tape drive is not allocated to another user
- Allocates the tape drive to you if the tape drive is available
- Verifies that the tape is placed and loaded on the device you specify

- Checks that the tape has been initialized, if you use a tape that has already been initialized
- Specifies the tape's density and format
- Verifies that the volume label on the tape (for ANSI-format tapes) matches the volume label you specify

If you are using an ANSI-formatted tape, you should specify the /FORMAT=ANSI qualifier with the MOUNT command; however, if you specify a volume label, this qualifier is the default. If you are not using an ANSI-formatted tape, you must specify the /FORMAT=FOREIGN qualifier with the MOUNT command. The /FORMAT=FOREIGN qualifier tells the systems that the tape does not have an ANSI format label. If you do not specify the MOUNT command, access to your tape is unrestricted. If you mount a tape /FORMAT=FOREIGN, no one else can access it.

See the *RSTS/E System User's Guide* for more information on the MOUNT command.

17.2.4.3 Example of Initializing a Tape on RSTS/E

The series of commands in the following example initialize the tape MYTAPE: on RSTS/E:

```
$ ALLOCATE MYTAPE:
$ INITIALIZE/FORMAT=ANSI MYTAPE: PRG
$ MOUNT MYTAPE: PRG
```

In this example:

1. The ALLOCATE command allocates the tape drive assigned to logical name MYTAPE:.
2. The INITIALIZE command initializes the tape, writes an ANSI-format volume label, and uses PRG as the volume label to identify the tape.
3. The MOUNT command specifies an ANSI file structure and PRG as the volume label.

To use a tape that has already been initialized, you only need to specify the ALLOCATE and MOUNT commands. For example:

```
$ ALLOCATE MYTAPE:
$ MOUNT/FORMAT=ANSI MYTAPE: PRG
```

In this example, the ALLOCATE command allocates the tape drive assigned to logical name MYTAPE: and tells the system that the tape on that tape drive is online and available for use, and verifies the volume label.

17.2.5 Opening a File on Tape

When you use a tape to handle and store records, you must open a file on it. To open a file on tape, you must use this format of the OPEN statement:

```
OPEN 'file-spec' [ FOR OUTPUT ] [ AS FILE ] [#]chnl-exp,
                 [ FOR INPUT  ]
ORGANIZATION SEQUENTIAL [(,clause), . . . ]
```

file-spec
Specifies the physical or logical name and number of the drive on which your tape is physically mounted and the file name and type of the file. Specifying a file name automatically accesses RMS–11.

FOR OUTPUT
Opens a new file.

FOR INPUT
Opens an existing file.

#chnl-exp
Is the number of the channel on which the file is open.

ORGANIZATION SEQUENTIAL
Specifies sequential file organization. You can also add the FIXED or VARIABLE keyword to the ORGANIZATION SEQUENTIAL clause, to indicate either a fixed-length or variable-length record, respectively.

clause
Can be any of the following clauses:

ACCESS
BLOCKSIZE int-exp
MAP mapname
NOREWIND
RECORDSIZE int-exp

The following sections describe the OPEN statement clauses. See Chapter 12 and the *BASIC–PLUS–2 Reference Manual* for more information on these clauses.

17.2.5.1 ACCESS Clause

The ACCESS clause determines how the program can use the file.

- ACCESS READ allows only FIND, GET, or other input statements on the file. The OPEN statement cannot create a file if the ACCESS READ clause is specified.
- ACCESS WRITE allows only PUT, UPDATE, or other output statements on the file.
- ACCESS MODIFY allows any I/O statement except SCRATCH on the file. ACCESS MODIFY is the default.
- ACCESS SCRATCH allows any I/O statement valid for a sequential or terminal-format file.
- ACCESS APPEND is the same as ACCESS WRITE for sequential files, except that BASIC–PLUS–2 positions the file pointer after the last record when it opens the file.

17.2.5.2 BLOCKSIZE Clause

The BLOCKSIZE clause specifies the physical size of a block on tape. The BLOCKSIZE value is the number of records in a physical block. The default is one record per block. You must specify the BLOCKSIZE value as an integer.

The size of the record buffer, in bytes, is the product of the values you specify in the RECORDSIZE and BLOCKSIZE clauses. For example, if your RECORDSIZE clause is 128 bytes and your BLOCKSIZE clause is 4, your record buffer size is 512 bytes. The total record buffer must be divisible by 4 and cannot exceed 8192 bytes. When you omit the BLOCKSIZE clause, the size of the record buffer is the value you specify in the RECORDSIZE clause.

17.2.5.3 MAP Clause

The MAP clause specifies the data type of the variables you list and allocates storage for them in the record buffer. The amount of storage allocated in a record buffer cannot exceed 8192 bytes. The MAP statement allocates permanent storage for your data, whereas the RECORDSIZE clause allocates dynamic storage and requires MOVE statements to move data in and out of a record buffer. If you use a MAP statement, you do not have to use a RECORDSIZE clause and vice versa.

17.2.5.4 RECORDSIZE Clause

The RECORDSIZE clause determines the record lengths for either all fixed-length records or the largest variable-length record. The maximum record length on a tape is 8192 bytes. The default length is 512 bytes. BASIC–PLUS–2 changes any RECORDSIZE value less than 18 to 18.

17.2.5.5 NOREWIND Clause

The NOREWIND clause controls the position of your tape. If you do not specify NOREWIND, the OPEN statement positions the tape at its beginning of tape (BOT), then searches for the file you specify until it reaches the logical end of tape. The logical end of tape is the position after the last record in the last file on tape. For example:

```
10     OPEN 'MM0:MYFILE.DAT' FOR OUTPUT AS FILE #4%,  &
                             ORGANIZATION SEQUENTIAL
```

This statement positions the tape at the beginning of the tape and opens the file, MYFILE.DAT, overwriting any files that may already exist on the tape. If you specify NOREWIND, the OPEN FOR OUTPUT statement positions the tape at the logical end of tape. For example:

```
10     OPEN 'MM0:MYFILE.DAT' FOR OUTPUT AS FILE #4%,   &
                             ORGANIZATION SEQUENTIAL,  &
                             NOREWIND
```

This statement opens the file MYFILE.DAT at the logical end of tape.

If you specify NOREWIND, the OPEN FOR INPUT statement searches for the file you specify without rewinding the tape to its beginning. If the file is not found, BASIC–PLUS–2 rewinds the tape to its beginning, then searches for the file again. For example:

```
10    OPEN 'MM0:MYFILE.DAT' FOR INPUT AS FILE #4%,   &
                            ORGANIZATION SEQUENTIAL
20    FOR CT% = 1% TO 50%
30    INPUT REC
40    MOVE TO #4%, REC
50    PUT #4%
60    NEXT CT%
70    CLOSE #4%
    .
    .
    .
1000 OPEN 'MM0:MYFILE.DAT' FOR INPUT AS FILE #4%, NOREWIND
```

1. Line 10 rewinds the tape, then searches for MYFILE.DAT.
2. Lines 20 through 60 perform 50 PUT operations.
3. The tape is positioned after the fiftieth record. Line 1000 does not rewind the tape before searching for MYFILE.DAT but starts searching at its current position to the logical end of tape. Because the file cannot be found from its current position (behind MYFILE.DAT), RMS rewinds the tape to the beginning of the tape and searches for MYFILE.DAT again.

By default, opening an RMS–11 file rewinds the tape. In summary, if you want to create a new file, use the NOREWIND clause with the OPEN FOR OUTPUT statement. If you think a file you want to access is located after the current tape position, use the NOREWIND clause with the OPEN FOR INPUT statement to search for your file and save time.

17.2.6 Writing Records to Tape

Each record is temporarily stored in a record buffer before being written to tape. If you use a MAP statement, this buffer or space is allocated at compile time and is permanent. If you use a RECORDSIZE clause, this buffer or space is allocated at run time and is dynamic.

Regardless of the method you use to determine the size of your record buffer, you must allocate a large enough buffer to accommodate the size of your records. If your buffer is too small and you try to write a record longer than the buffer, BASIC–PLUS–2 signals the error message "Magtape record length error." If your buffer is too large, you waste unused space. You can use the COUNT clause with the PUT statement to specify record lengths that are smaller than the buffer size you specify.

After a file is open, you use the PUT statement to write records to tape. The format of the PUT statement is as follows:

```
PUT #chnl-exp [,COUNT int-exp]
```

chnl-exp
Is the channel number of the open file.

COUNT int-exp
Specifies the record's size. The record size value must be an integer. If you use the FIXED clause, the COUNT clause serves no purpose.

For example:

```
100 OPEN 'TAPE:PARTS.DAT' FOR OUTPUT AS FILE #2%,   &
            ORGANIZATION SEQUENTIAL VARIABLE,       &
            RECORDSIZE 128,                         &
            BLOCKSIZE 4
    DECLARE DOUBLE PART_CODE,                       &
            STRING PART_NAME
    ASK: INPUT 'Part code';PART_CODE
         INPUT 'Part name';PART_NAME
         MOVE TO #2%, PART_CODE, PART_NAME
         PUT #2%
         INPUT 'Write another record <YES OR NO>';ANS$
         IF ANS$ = 'YES'
           THEN GOTO ASK
           ELSE CLOSE #2%
         END IF
32767    END
```

In this example:

1. The OPEN statement opens the file PARTS.DAT for output on the tape drive assigned to the logical name TAPE:.

   The maximum record size is 128 bytes and there are 4 of these records to a block. The total size of the record buffer is 512 bytes. Every fourth PUT causes a physical write, moving 512 bytes of data to tape.

2. The MOVE statement moves the record to the record buffer.
3. The PUT statement moves the record from the record buffer and writes it to tape.

17.2.7 Adding New Records to Tape

You can add records to a tape only after the last record in the last file on a tape (logical end of tape). For example, if you open FILE1.DAT and write 50 records to it, and it is the only file written to the tape, when you close the file, RMS–11 writes an end of tape mark after the last record.

If you want to add records to the file, you must position the current record pointer to the end of a file by adding the ACCESS APPEND clause to the OPEN statement. If you do not use ACCESS APPEND, the current record pointer points to the first record in the file and the first PUT operation overwrites the first record. Successive PUT operations overwrite successive records.

```
100 MAP (EX) DOUBLE PART_CODE,    &
             STRING PART_NAME,    &
             WORD FILL(52)
    OPEN 'MM1:PARTS.DAT' AS FILE #2%,          &
             ORGANIZATION SEQUENTIAL VARIABLE, &
             ACCESS APPEND,                    &
             MAP EX
    ASK: INPUT 'Part code';PART_CODE
         INPUT 'Part name';PART_NAME
         PUT #2%
         INPUT 'Write another record <YES OR NO>';ANS$
         IF ANS$ = 'YES'
           THEN GOTO ASK
           ELSE CLOSE #2%
         END IF
32767    END
```

In this example:

1. The MAP statement allocates permanent storage for a DOUBLE floating-point variable (8 bytes) and a STRING variable (16 bytes). The FILL statement pads the unused buffer space (52 WORDs or 104 bytes).
2. The OPEN statement opens the existing file PARTS.DAT stored on the tape that is physically mounted on tape drive MM1:. ACCESS APPEND positions the current record pointer at the logical end of the file. The MAP clause references the MAP statement in line 100.
3. The INPUT statements prompt you to enter a record and assign your responses to the variables PART_CODE and PART_NAME.
4. The PUT statement writes the record to tape.
5. The next INPUT statement asks you if you want to add another record. The IF statement evaluates your response. If you answer YES, control branches to label ASK. If you do not answer YES, the file is closed.

17.2.8 Reading Records on Tape

You can use the GET statement to retrieve records you wrote to a tape. The format of the GET statement is as follows:

GET #chnl-exp

#chnl-exp
Is the channel number of the open file.

The following program opens an existing file and reads the number of records you specify:

```
1   ON ERROR GOTO 19000
    MAP (PARTS) DOUBLE PART_NUM,    &
                STRING PART_NAM,    &
                BYTE FILL(104)
100 OPEN 'MYTAPE:PARTS.DAT' FOR INPUT AS FILE #3%,     &
                ORGANIZATION SEQUENTIAL VARIABLE,      &
                ACCESS READ,                           &
                NOREWIND,                              &
                MAP PARTS
   ASK:  INPUT 'How many records do you want to read';RET_NUM%
   LOOP: FOR MYLOOP% = 1% TO RET_NUM%
            GET #3%
            PRINT 'The product name is', PART_NAM
            PRINT 'The part number is', PART_NUM
         NEXT MYLOOP%
         GOTO 32766
19000    IF (ERR = 11%)
           THEN
              PRINT 'You have reached the end of the file.'
              RESUME 32766
           ELSE
              ON ERROR GOTO 0
         END IF
32766    CLOSE #3%
32767    END
```

In this example:

1. Line 1 transfers control to an error handler starting on line 19000 if an error occurs.
2. The MAP statement declares the storage for the variables listed.
3. The OPEN statement opens the file PARTS.DAT for input on the tape drive assigned to the logical name MYTAPE:.
4. LOOP: labels a loop to read the number of records you want, assigns them to the variables declared in the MAP statement, and prints the values of the variables.
5. If the error is "End of file on device" (ERR=11), the error handler at line 19000 displays the message, "You have reached the end of the file". If another error is signaled, error handling returns to BASIC–PLUS–2.

17.2.9 Locating Records on Tape

If you are familiar with the contents of your file, you can locate and read the record you want with FIND and GET statements. The FIND statement sets the current record pointer, and if you use a GET statement, you can then read the record at the current record pointer. The format of the FIND statement is as follows:

FIND #chnl-exp

#chnl-exp
Is the channel number of the open file. It must be preceded by a number sign (#).

FIND operations perform faster than GET operations because no data is transferred. For example:

```
1       ON ERROR GOTO 19000
    MAP (PARTS) DOUBLE PART_CODE,    &
                STRING PART_NAME,    &
                BYTE FILL(104)
    OPEN 'MM1:PARTS.DAT' FOR INPUT AS FILE #4%,    &
                ORGANIZATION SEQUENTIAL VARIABLE,  &
                ACCESS READ,                       &
                NOREWIND,                          &
                MAP PARTS
   LOOP: FIND #4% FOR MYLOOP% = 1% TO 50%
         GET #4%
10  ASK: PRINT 'This is the current record: '; PART_NAME, PART_CODE
         PRINT 'Do you want to move the Current Record Pointer'
         INPUT 'another fifty records';ANS$
         CONV$ = EDIT$(ANS$,32%)
         IF CONV$ = 'YES'
               THEN GOTO LOOP
               ELSE GOTO 32766
         END IF
19000    IF (ERR = 11%)
               THEN PRINT 'You have reached the end of the file.'
               RESUME 10
         END IF
32766    CLOSE #3%
32767    END
```

In this example:

1. Line 1 transfers control to line 19000 if an error occurs.
2. The MAP statement declares the storage for three string variables.

3. The OPEN statement opens a file on the tape drive MM1:. The NOREWIND clause tells RMS not to rewind the tape to its beginning (BOT) before searching for PARTS.DAT.
4. The FIND statement sets the current record pointer 50 records away from its current position.
5. The GET statement retrieves that record and assigns it to the variables declared in the MAP statement.
6. The PRINT statement displays the contents of the record.
7. The INPUT statement asks you if you want to move the current record pointer another 50 records away from its current position. If you answer YES, control passes to label LOOP:, which moves the current record pointer another 50 records.
8. If BASIC–PLUS–2 signals the error "End of file on device" (ERR=11), control branches to label ASK:. If another error occurs, control branches to the CLOSE statement at line 32766.

17.2.10 Truncating Files on Tape

You can truncate the records in a file by using the ACCESS SCRATCH clause in the OPEN statement. This clause lets you delete all the records from the current record to the end of the file, including the record at the current record pointer. For example:

```
10      ON ERROR GOTO 19000
        !
        DECLARE WORD CT
        !
        MAP (MYMAP) WORD A
        !
        OPEN 'XX:PARTS.DAT' FOR OUTPUT AS FILE #3%,        &
                ORGANIZATION SEQUENTIAL,                   &
                MAP MYMAP,                                 &
                BLOCKSIZE 10%
        !
        ! Loop to write 10 records
        !
                FOR CT = 1% TO 10%
                        A = CT
                        PUT #3%
                NEXT CT
        !
        CLOSE #3%
```

```
        !
        ! Open the file with ACCESS SCRATCH for truncating records
        !
        OPEN 'XX:PARTS.DAT' FOR INPUT AS FILE #3%,        &
                ORGANIZATION SEQUENTIAL,
                MAP MYMAP,                                &
                BLOCKSIZE 10%,                            &
                ACCESS SCRATCH
        !
        ! Loop to read five records
        !
                FOR CT = 1% TO 5%
                         GET #3%
                NEXT CT
        !
        ! Delete records six through ten
        !
        SCRATCH #3%
        !
        ! Write 10 records
        !
                FOR CT = 11% TO 20%
                        A = CT
                        PUT #3%
                NEXT CT
        !
        OPEN 'XX:PARTS.DAT' FOR INPUT AS FILE #3%,        &
                ORGANIZATION SEQUENTIAL,                  &
                MAP MYMAP,                                &
                BLOCKSIZE 10%
        !
        ! Loop to read 15 records
        !
                FOR CT = 1% TO 15%
                        GET #3%
                        PRINT A
                NEXT CT
        !
        GOTO 32766
        !
19000   ! Error Handler
        !
        PRINT 'Unexpected error: ';ERT$(ERR)
        RESUME 32766
        !
32766   CLOSE #3%
32767   END
```

In this example:

1. The first OPEN statement creates a new file to write records.

2. The first FOR...NEXT loop writes 10 records.
3. The second OPEN statement opens the file with the ACCESS SCRATCH clause.
4. The second FOR...NEXT loop reads five records. The current record position is at the beginning of the sixth record.
5. The SCRATCH statement truncates all records from the current record to the end of the file.
6. The current record pointer is positioned after the fifth record. The third FOR...NEXT loop writes 10 new records.
7. The third OPEN statement opens the file to read records.
8. The fourth FOR...NEXT loop retrieves and displays the remaining 15 records.

17.2.11 Dismounting a Tape

To dismount a tape, you must do the following:

- If you specified the MOUNT command, specify the DCL command DISMOUNT
- Specify the DCL command DEALLOCATE
- Physically dismount the tape

For example:

```
$ DISMOUNT MYTAPE:
$ DEALLOCATE MYTAPE:
```

You must specify the DISMOUNT command before you do any of the following:

- Issue the DEALLOCATE command
- Set the tape drive offline
- Rewind, unload, or take the tape off the drive

The DEALLOCATE command releases the tape drive you allocated and allows other users to access that device. This command applies only to systems that have multiuser protection. If you are a nonprivileged user, you can deallocate only the tape drive you allocated. When you log out, the system automatically deallocates all tape drives allocated to you.

After you deallocate the tape drive, press the UNLOAD button on the tape drive to unload and rewind your tape.

17.2.12 Closing a File on Tape

You should explicitly close every file you open. The CLOSE statement ends I/O operations to an open file. The format of the CLOSE statement is as follows:

```
CLOSE [#]chnl-exp, . . .
```

chnl-exp
Is the number of the channel on which the file is open. It can be preceded by an optional number sign (#).

If the file is open FOR INPUT, CLOSE closes the open file. If the file is open FOR OUTPUT, BASIC–PLUS–2 performs the following operations:

- Writes a file trailer label (one end-of-file mark) following the last record
- Backspaces over the last end-of-file mark
- Releases the space allocated for the record buffer

The CLOSE statement does not rewind a tape. You must press the rewind button on your tape drive to rewind it.

17.3 Device-Specific I/O

Device-specific I/O lets you perform I/O directly to a device.

The advantages of device-specific I/O are as follows:

- Your program collects data faster than using RMS–11.
- Your program does not need to link to RMS–11 libraries; therefore, you save at least 8K words of virtual address space for your task.

Note

When doing device-specific I/O, all device names must be specified in uppercase letters.

The following sections describe device-specific I/O to unit record devices, tapes, and disks.

17.3.1 Device-Specific I/O to Unit Record Devices

You perform device-specific I/O to unit record devices by using only the device name in the OPEN statement file specification. You should allocate the device at DCL command level before reading or writing to the device. For example, this command allocates a card reader:

```
$ ALLOCATE CR1:
```

Once the device is allocated, you can read records from it.

Example

```
MAP (DNG) A% = 80%
OPEN "CR1:" FOR INPUT AS FILE #1%, MAP DNG
GET #1%
```

BASIC–PLUS–2 treats the device as a file, and data is read from the card reader as a series of fixed-length records.

17.3.2 Device-Specific I/O to Magnetic Tape Devices

When performing device-specific I/O to a tape drive, you open the physical device and transfer data between the tape and your program. GET and PUT statements perform read and write operations. UPDATE and DELETE statements are invalid when you perform device-specific I/O.

BASIC language elements allow you to do the following:

- Identify the physical attributes of the tape, including:
  - Recording density
  - 7-track or 9-track
  - Parity
- Control the physical movement of the tape, including:
  - Writing end-of-file marks to tape
  - Physically moving the tape to access blocks and records
  - Rewinding the tape
- Define the I/O buffer size, record size, and block size
- Define the size of each record in a file

If you do not use RMS–11, the Task Builder does not need to link to RMS–11 libraries. Therefore, add the /NOSEQUENTIAL qualifier to the BUILD command to generate Task Builder command and overlay descriptor language files that do not access RMS–11 libraries. By specifying the /NOSEQUENTIAL

qualifier with the BUILD command, you free 8K words of virtual address space and substantially decrease the time it takes to link your program.

17.3.2.1 Allocating and Mounting a Tape

You must allocate the tape unit to your process before starting file operations. For example, the following command line assigns tape drive MM1: to your process.

```
$ ALLOCATE MM1:
```

On RSX systems, specify the DCL command MOUNT with the /FOREIGN qualifier to mount the tape. For example:

```
$ MOUNT/FOREIGN MM1:
```

On RSTS/E systems, specify the DCL command MOUNT with the /FORMAT=FOREIGN qualifier to mount the tape. For example:

```
$ MOUNT/FORMAT=FOREIGN MM1:
```

You must specify these qualifiers with the MOUNT command because otherwise the file structure is unknown to the operating system. Without these qualifiers, the operating system attempts to read and interpret a file format label that does not exist.

Once you prepare a tape, you can use the following BASIC–PLUS–2 statements to handle file operations:

- OPEN
- PUT
- GET
- MAP
- MOVE
- MAP DYNAMIC
- REMAP
- CLOSE

17.3.2.2 Opening a File on Tape

To open a channel to a tape, you must use the following format of the OPEN statement:

```
OPEN 'dev:' [ FOR OUTPUT ] AS FILE [ #]chnl-exp [,clause, . . . ]
            [ FOR INPUT  ]
```

dev:
Is the physical or logical name of the tape drive on which your tape is physically mounted.

FOR OUTPUT
Opens a tape file for writing records only.

FOR INPUT
Opens a tape file.

chnl-exp
Is the number of the channel on which the tape is open. You can precede the *chnl-exp* with an optional number sign (#).

[,clause]
Specifies any of the following clauses to the OPEN statement. These are the only clauses valid for device-specific I/O.

MAP mapname
RECORDSIZE int-exp
MODE int-exp

For device-specific I/O, the default record size is 512. If you specify a record size less than 512, BASIC–PLUS–2 opens the device with a record size of 512 and does not signal an error. If you allocate a static buffer with a MAP clause that is less than 512, BASIC–PLUS–2 signals the error "Bad RECORDSIZE value on OPEN" (ERR=148).

The MODE clause sets the density and parity of your tape. You determine the value of MODE with the following formula:

MODE = E + P

E Is the recording density. Its value can be 256, which sets the recording density to 1600 bits per inch (BPI) with phase encoding, or 1, which sets the recording density to the system default.

P Is the parity check to be performed. Its value can be 0 to check for odd parity or 1 to check for even parity.

Table 17–1 lists the MODE values and describes the functions those values perform.

Table 17–1 MODE Values

E	P	MODE Value	Meaning
1	0	1	Sets the recording density to the default, enables phase encoding, and checks for odd parity.
1	1	2	Sets the recording density to the default, enables phase encoding, and checks for even parity.
256	0	256	Sets the recording density to 1600 bits per inch (BPI) and checks for odd parity.
256	1	257	Sets the recording density to 1600 BPI and checks for even parity.

Parity checking is the process of verifying whether or not a bit has been dropped. BASIC–PLUS–2 does this by counting the total number of bits set for a byte. If you choose a value of 0 (for odd parity), BASIC–PLUS–2 checks if the total number of bits set for each byte is odd. If the total number is even, BASIC–PLUS–2 displays an error message.

The most important consideration is the tape drive's default for the recording density. You cannot specify 1600 BPI if your tape drive is only equipped to record at 800 BPI.

In the following example, line 10 sets the recording density to 1600 BPI with odd parity checking:

```
10 OPEN 'MM0:' FOR OUTPUT AS FILE #9, MODE 256
```

17.3.2.3 Opening a Tape File for Output

The following statement opens the tape drive MT1: for writing:

```
OPEN "MT1:" FOR OUTPUT AS FILE #1%
```

17.3.2.4 Opening a Tape File for Input

The following statement opens the tape unit MT2: for input:

```
OPEN "MT2:" AS FILE #2%
```

Depending on how you access records, there are two ways to open a foreign magnetic tape. If your program uses dynamic buffering and MOVE statements, open the file with no RECORDSIZE clause.

When processing records, each GET operation will read one physical record whose size is returned in RECOUNT. If you are using a map only, the first *n* bytes (where *n* is the value returned in RECOUNT) are valid.

17.3.2.5 Closing a File on Tape

The CLOSE statement ends I/O to the tape. For example, the following statement ends input and output to the tape open on channel #12.

```
CLOSE #12%
```

The CLOSE statement releases the allocated record buffer space and closes the file.

BASIC–PLUS–2 does not write any end-of-file marks for device-specific I/O to magnetic tape. Use the MAGTAPE function to explicitly write end-of-file marks for magnetic tape.

The CLOSE statement does not rewind your tape unless you specify the RESTORE statement in your program.

17.3.2.6 Using the MAGTAPE Function

You can include the MAGTAPE function in your program to perform the following operations:

- Rewind the tape and set the tape drive off-line
- Rewind the tape without setting the tape drive off-line
- Write an end-of-file mark at the current position of the tape
- Skip records
- Backspace records
- Set the density and parity of a tape
- Monitor the status of I/O to a tape

The format of the MAGTAPE function is as follows:

int-var1 = MAGTAPE(func-code, int-var2, chnl-exp)

int-var1
Is a value returned by function codes 4, 5, and 7. See Table 17–2 for the values of int-var1.

func-code
Is an integer from 1 through 7 that specifies the code for the MAGTAPE function you want to perform. See Table 17–2 for a list of the function codes and what they do.

int-var2
Is an integer parameter for function codes 4, 5, and 6. See Table 17–2 for values of int-var2.

chnl-exp
Is the channel number the tape is opened on.

Table 17–2 MAGTAPE Function Codes

Function Code	Parameter	Value Returned	Explanation
1	Unused	0	Rewinds the tape open on the channel number you specify and sets the tape drive on which it is physically mounted off-line. For example: `200 I% = MAGTAPE(1%,0%,2%)` This example rewinds the tape open on channel 2 and sets it off-line.

2	Unused	0	Writes one end-of-file mark at the current position of the tape. Use this function twice after the last record to indicate the end of tape. For example:
			`200 I% = MAGTAPE(2%,0%,2%) !write out two end-of-file's`
			`200 I% = MAGTAPE(2%,0%,2%)`
			This example writes an end-of-file to the record at the current position of the tape open on channel 2.

3	Unused	0	Rewinds the tape open on the channel you specify to its BOT. For example: `200 I% = MAGTAPE(3%,0%,2%)` This example rewinds the tape open on channel 2, but does not take the tape off-line.

(continued on next page)

Table 17–2 (Cont.) MAGTAPE Function Codes

Function Code	Parameter	Value Returned	Explanation
4	Number of records to skip (1 to 32767)	0, or number of records not skipped	Advances the tape until either the number of records you specify is skipped, or the tape reaches the end of the file. If the tape reaches the end of file, the value returned (I%) equals the number you specified (P%) minus the number of records actually skipped before reaching the end of the file. For example: `200 I% = MAGTAPE(4%,50%,2%)` This example skips 50 records on the tape open on channel #2. If the search does not reach the end of the file, the value of I% is 0. If the end of the file is reached after skipping 30 records, the value of I% is 20.
5	Number of records to backspace (1 to 32767)	0, or number or records not backspaced	Backspaces the length of a record for the number of records you specify until the number of records you specify to backspace is reached, or until a BOT or end-of-file is reached. For example: `200 I% = MAGTAPE(5%,1%,2%)` This example backspaces the length of one record on the tape that is open on channel 2. If the tape is not at the BOT, the value of 1% is 0%. If the tape is at the BOT, the value of 1% is 1%.
6	E+D*4+P	0	Sets the density and parity of your tape. E is the recording density. Its value can be 256, which sets the recording density to 1600 BPI or 0. If the value is 0, the recording density defaults to one of the values for D. D is a value that indicates the recording density and type of track. You can choose one of these values:

Value	Meaning
0	200 BPI (7-track only)
1	556 BPI (7-track only)
2	800 BPI (7-track only)
3	800 BPI (9-track only)

(continued on next page)

Table 17–2 (Cont.) MAGTAPE Function Codes

Function Code	Parameter	Value Returned	Explanation

			P is the parity check to be performed. Its value can be 0 for odd parity checking or 1 for even parity checking.
			For example:
			`10 OPEN "MMO:" AS FILE 2%`
			`20  I%  =  MAGTAPE(6%, 0%+2%*4%+1%, 2%)`
			In this example, reading the MAGTAPE function from left to right, 6% is the number of the channel this tape is open on. 0% is the value of E, which defaults to one of the values of D. 2% is the value of D, which sets the BPI to 800 and specifies the tape to be 7-track. 4% is a numeric constant. 1% is the value of P, which checks for even parity. 2% is the number of the channel that this tape is open on.

7	Unused	Status word	Returns a WORD integer representing the status of the I/O operations your programs perform to tape. Each I/O operation sets bits in a certain sequence. This MAGTAPE function code returns a value depending on how these bits are set. To find out what the value of a particular bit is, you must include a test in your program. The value returned as a result of the test corresponds to an I/O operation. See Table 17–3 for a description of the bits set, the logical test you must include to evaluate the returned value, and what the logical test value means.
8†	Unused	Characteristic word	Returns a WORD integer representing the file characteristics of the specified magnetic tape.
9†	Unused	0	Rewinds the tape once the file is closed.

†RSTS/E systems only.

If the specified function code is 7, *int-var1* is a 16-bit integer that reflects the status of the specified magnetic tape. Table 17–3 provides information on the bit values and their meanings.

Table 17–3 Tape Status Word

Bit Set	Status Word	Logical Test	Explanation
15	–32768	I% < 0%	Last command caused an error.
14–13	24376	(I% AND 24576%)/8192 = 0	Recording density is 200 BPI.
		(I% AND 24576%)/81925 = 1	Recording density is 556 BPI.
		(I% AND 24576%)/81925 = 2	Recording density is 800 BPI and the tape is 7-track.
		(I% AND 24576%)/81925 = 3	Recording density is 800 BPI and the tape is 9-track.
12	4096	(I% AND 4096%) = 0%	Tape is 9-track.
		(I% AND 4096%)<>0%	Tape is 7-track.
11	2048	(I% AND 2048%) = 0%	Odd parity checking.
		(I% AND 2048%)<>0%	Even parity checking.
10	1024	(I% AND 1024%)<>0%	Tape is physically write-locked.
9	512	(I% AND 512%)<>0%	Tape is past the EOT marker.
8	256	(I% AND 256%)<>0%	Tape is at the BOT.
7	128	(I% AND 128%)<>0%	Last command detected an end-of-file.
6	None.	None.	Reserved.
5	32	(I% AND 32%)<>0%	Tape drive is off-line.
4	16	(I% AND 16%)<>0%	Tape drive is one of the following: TU16, TE16, TU45, or TU77.
3	8	(I% AND 8%)<>0%	BPI is 1600 and phase-encoded.
2–0	None.	None.	Reserved.

17.3.2.7 Writing Records to Tape

The PUT statement writes records to a tape, and successive PUT operations write successive records. If you do not use MAP statements, you must use the RECORDSIZE clause to define the size of your record buffer and you must use MOVE statements to transfer records in and out of the record buffer. For example:

```
10    OPEN 'MM0:' FOR OUTPUT AS FILE #9%, RECORDSIZE 512%
      I% = MAGTAPE(3%,0%,9%)
      I% = MAGTAPE(7%,0%,9%)
```

```
      IF ((I% AND 256%) <> 0%)
        THEN
           PRINT 'The tape is at its BOT.'
           INPUT 'What is the temperature in Celsius'; CEL_TEMP
           MOVE TO #9%, CEL_TEMP
           PUT #9%
           I% = MAGTAPE (2%,0%,9%)
           I% = MAGTAPE (2%,0%,9%)
           I% = MAGTAPE (3%,0%,9%)
        ELSE
           PRINT 'The tape is not at its BOT.'
      END IF
      CLOSE #9%
32767 END
```

In this example:

1. The OPEN statement opens a tape physically mounted on the tape drive MM0: for output, and specifies a maximum record size of 512 bytes.
2. The first MAGTAPE function rewinds the tape open on channel 9 to its BOT.
3. The second MAGTAPE function returns a status variable and the IF statement performs a logical test to find out if the tape has been rewound to its BOT (see Table 17–3 for the logical test).
4. The INPUT statement prompts you for the value of the variable CEL_TEMP.
5. The MOVE statement moves the value of CEL_TEMP into the record buffer.
6. The PUT statement moves the data from the record buffer and writes it to tape.
7. The next two MAGTAPE functions write end-of-file marks, indicating the end of the tape.
8. The last MAGTAPE function rewinds the tape.

17.3.2.8 Adding New Records to Tape

To add records to an existing file on tape, you must position the tape to the end of the file. You cannot use the ACCESS APPEND clause to do this because you are not using RMS–11. If you do not position your tape to its end-of-file, any new records you write overwrite existing records. Therefore, you must use the MAGTAPE function to skip enough records to position the tape at the end of the file. For example:

```
10       OPEN 'MM1:' AS FILE #2%, RECORDSIZE 512
  START: INPUT 'Number of records to skip';SKIP%
         I% = MAGTAPE(4%,SKIP%,2%)
         PRINT 'You have skipped';I%;' records.'
         I% = MAGTAPE(7%,0%,2%)
         IF (I% AND 128%) <> 0%
           THEN
              PRINT 'The last statement detected an end of file mark.'
              INPUT 'Record';RECORD$
              MOVE TO #2%, RECORD$
              PUT #2%
           ELSE
              PRINT 'You are not at the end of the file.'
              GOTO START
         END IF
         I% = MAGTAPE(2%,0%,2%)
32766    CLOSE #2%
32767    END
```

In this example:

1. The OPEN statement opens a channel to the tape physically mounted on MM1:.
2. The INPUT statement prompts you to enter the number of records to skip.
3. The first MAGTAPE function skips the number of records you specify, and the PRINT statement displays the number of records the operation was able to skip.
4. The IF statement tests to see if the status value returned by the second MAGTAPE function indicates that the current tape position is at the end of the file. If the test is true, a message displays, you are prompted to input a record, the MOVE statement transfers the record to the record buffer, then the PUT statement writes the record to tape. If the test is not true, a message displays that you are not at the end of the file, and control passes to the label START.
5. The last MAGTAPE function writes a new end-of-file mark at the end of the record just added.

17.3.2.9 Reading Records on Tape

The GET statement reads records from an open tape and successive GETs read successive records. When you open an existing file FOR INPUT, you must use the same record size you used to write the record. For instance, this example gives you an error:

```
10      OPEN 'MY:' FOR OUTPUT AS FILE #5%, RECORDSIZE 512%
20      INPUT REC
30      MOVE TO #5%, REC
40      PUT #5%
50      CLOSE #5%
        .
        .
        .
500     OPEN 'MY:' FOR INPUT AS FILE #4%, RECORDSIZE 128%
```

The RECORDSIZE clause in the OPEN statement at line 10 defines a record size of 512 bytes. When the program tries to open the same file at line 500, the stated record size is 128 bytes, so BASIC–PLUS–2 will return an error message.

```
1           ON ERROR GOTO 19000
            OPEN 'MM1:' FOR INPUT AS FILE #1%, RECORDSIZE 512

  LOOP: FOR MYCOUNT% = 1% TO 128%
           GET #1%
           MOVE FROM #1%, REC
           PRINT REC
        NEXT MYCOUNT%
        I% = MAGTAPE(3%,0%,1%)
        GOTO 32766
19000   IF (ERR = 11%)
           THEN PRINT 'You have reached the end of the file.'
           ELSE PRINT ERT$(ERR)
        END IF
        RESUME 32766
32766   CLOSE #1%
32767   END
```

In this example:

1. Line 1 transfers control to line 19000 if an error occurs.
2. The OPEN statement opens a tape on channel #1% for input and declares the record buffer size to be 512 bytes.
3. Label LOOP: starts a loop to retrieve 128 records and print them.
4. The MAGTAPE function will rewind the tape after line 32766 executes.

5. The error handler at line 19000 prints a message if the end of the file is reached. If another error occurs, the message associated with the error displays.

17.3.2.10 Locating Records on Tape

To locate records, you must know the contents of the tape. Then you use the MAGTAPE function to skip or backspace the number of records to find the one you want. For example:

```
10      ON ERROR GOTO 19000
        !
        DECLARE WORD CT
        !
        MAP (FUN) STRING RECORD1 = 128%, &
                  STRING RECORD2 = 128%, &
                  STRING RECORD3 = 128%, &
                  STRING RECORD4 = 128%
        !
        OPEN 'XX:' FOR INPUT AS FILE #3%, MAP FUN
        !
        ! Rewind tape to its BOT
        !
        I% = MAGTAPE(3%,0%,3%)
        !
 Choice: PRINT 'Do you want to: 1 = move forward, '
        !
        INPUT '2 = move backward, or 3 = read records';ANS%
        !
        SELECT ANS%
                CASE = 1%
                        INPUT 'Records to skip';ANS1%
                        !
                        ! Skip records
                        !
                        I% = MAGTAPE(4%,ANS1%,3%)
                        PRINT 'Skipping';ANS1%;' records.'
                        GOTO Choice
                        !
                CASE = 2%
                        INPUT 'Records to backspace';ANS2%
                        !
                        ! Backspace records
                        !
                        I% = MAGTAPE(5%,ANS2%,3%)
                        PRINT 'Back tracking';ANS2%;' records.'
                        GOTO Choice
```

```
                CASE = 3%
                        INPUT 'Records to read';ANS3%
                        !
                        FOR CT = 1% TO ANS3%
                               GET #3%
                               PRINT RECORD1, RECORD2, RECORD3, RECORD4
                        NEXT CT
                        !
                        ! Check for the end-of-file mark
                        !
                        I% = MAGTAPE(7%,0%,3%)
                        PRINT 'Checking for end of file.'
                        !
                        ! Test for the end-of-file mark
                        !
                        IF (I% AND 128%) <> 0
                            THEN
                                 PRINT 'Tape is at the end of the file.'
                        END IF
                 CASE ELSE
                        PRINT 'Invalid number.  Try again.'
                        GOTO Choice
        END SELECT
        !
        GOTO 32766
        !
19000   ! Error Handler
        !
        PRINT 'Unexpected error: ';ERT$(ERR)
        RESUME 32766
        !
32766   CLOSE #3%
32767   END
```

In this example:

1. The MAP statement allocates storage for four variables.
2. The OPEN statement opens the tape for input on channel 3.
3. The first MAGTAPE function rewinds the tape to its BOT.
4. The INPUT statement asks you to choose whether you want to skip, backtrack, or read records.
5. If you answer 1, all the statements in the first CASE statement execute. The INPUT statement asks you for the number of records to skip. Your response is assigned to a variable that is used as the parameter in the MAGTAPE function.

6. If you answer 2, all the statements in the second CASE statement execute. The INPUT statement asks you for the number of records to backspace. Your response is assigned to a variable that is used as the parameter in the MAGTAPE function.

7. If you answer 3, all the statements in the third CASE statement execute. The INPUT statement asks you for the number of records to read. Your response is assigned to a variable that is used in a loop to retrieve and display records.

8. If you choose an invalid selection number, a message is displayed and control transfers to the label Choice.

9. The last MAGTAPE function compares the values of the MAGTAPE function and the tape status word to check for the end of the file.

17.3.2.11 Deleting Records on Tape

You can truncate the remaining records in a file using the MAGTAPE function to write an end-of-file mark to the record at the current position of the tape. For example:

```
10    MAP (BUF) INTEGER LONG QKIT_NUM, LONG FILL(127)
      OPEN 'MM2:' FOR INPUT AS FILE #4%, MAP BUF
      I% = MAGTAPE(4%,27%,4%)
      GET #4%
      PRINT QKIT_NUM
      INPUT 'Do you want to delete the remaining records <YES/NO>';ANS$
      IF ANS$ = 'NO'
         THEN I% = MAGTAPE(3%,0%,4%)
         ELSE I% = MAGTAPE(2%,0%,4%)
      END IF
32766 CLOSE #4%
32767 END
```

In this example:

1. The MAP statement specifies a record buffer of 512 bytes.

2. The OPEN statement opens a tape for input on channel 4.

3. The first MAGTAPE function tells the tape to skip the length of 27 records, which positions the tape at the beginning of the 28th record.

4. The GET statement gets the record at the current position of the tape.

5. The PRINT statement displays the record.

6. The IF statement asks you if you want to truncate the remaining records. If you do not want to truncate the remaining records, the next MAGTAPE function will rewind the tape. If you do want to truncate the remaining records, the last MAGTAPE function writes an end-of-file mark to the current tape position.

17.3.2.12 Dismounting a Tape

After you finish using a tape, you must do the following:

- Specify the DCL command DISMOUNT
- Specify the DCL command DEALLOCATE
- Physically dismount the tape

For example:

```
$ DISMOUNT MYTAPE:
$ DEALLOCATE MYTAPE:
```

You must specify the DISMOUNT command before you do any of the following:

- Issue the DEALLOCATE command
- Set the tape drive off line
- Rewind, unload, or take the tape off the drive

The DEALLOCATE command releases the tape drive you allocated and allows other users to access that device. This command applies only to systems that have multiuser protection. If you are a nonprivileged user, you can deallocate only the tape drive you allocated. When you log out, the system automatically deallocates all tape drives allocated to you.

After you deallocate the tape drive, press the UNLOAD button on the tape drive to unload and rewind your tape.

17.3.3 Device-Specific I/O to Disk Devices

When performing device-specific I/O to disks, you write and read data with PUT and GET statements. Note that, when accessing disks with device-specific I/O operations, you are performing logical I/O. Because of this, you should be careful not to overwrite block number zero, which is often the disk's boot block.

When you write records to disk, you can store as many records as can fit in a 512-byte disk block. The maximum number of blocks on a disk depends on the type of disk you use. You specify the location and format of the records in each block by blocking and deblocking records with MAP, MAP DYNAMIC, REMAP, and MOVE statements.

When you do not use RMS, the Task Builder does not need to link to RMS–11 libraries. Therefore, add the /NOVIRTUAL qualifier to the BUILD command to generate Task Builder CMD and ODL files that do not access RMS–11 libraries. By specifying the /NOVIRTUAL qualifier, you free 8K words of virtual address space and substantially decrease the time it takes to link your program.

You can use the following statements to handle file operations on disk:

- OPEN
- PUT
- GET
- CLOSE

The following sections describe device-specific I/O to disks.

17.3.3.1 Allocating and Mounting a Disk

You must allocate a disk unit to your process before starting operations. For example, the following command line assigns disk DU2: to your process.

```
$ ALLOCATE DU2:
```

On RSX systems, when you perform I/O directly to a disk, you must mount the disk with the MOUNT command and the /FOREIGN qualifier before accessing it. For example:

```
$ MOUNT/FOREIGN DU2:
```

You can then open the disk for input or output.

17.3.3.2 Opening a File on Disk

You open a channel to a disk using the OPEN statement with the ORGANIZATION VIRTUAL clause. The format of the OPEN statement is as follows:

OPEN 'dev:' AS FILE [#]chnl-exp ORGANIZATION VIRTUAL [, clause, . . .]

dev:
Is the physical name and number or logical name of the drive on which your disk is placed. Note that you do not include a file name.

chnl-exp
Is the number of the channel on which the disk is open. It can be preceded by an optional number sign (#).

clause
Is the RECORDSIZE or MAP clause.

Once you finish file operations, you should explicitly close all open files with the CLOSE statement. When BASIC–PLUS–2 executes the CLOSE statement, it releases any allocated record buffer space.

The following example opens the disk DU2: for output:

```
OPEN "DU2:" FOR OUTPUT AS FILE #2%, ORGANIZATION VIRTUAL  &
            RECORDSIZE 512
```

You can then write data to the disk.

The record size determined by the MAP or RECORDSIZE clause must be an integer multiple of 512 bytes.

17.3.3.3 Writing Records to Disk

You use the PUT statement to write records to a disk. Each PUT operation writes an integral number of 512-byte blocks. You can write records either randomly by block number or sequentially by omitting the RECORD clause to the PUT statement.

The format of the PUT statement is as follows:

PUT [#]chnl-exp ,RECORD int-exp

chnl-exp
Is the number of the channel on which the disk is open. It can be preceded by an optional number sign (#).

RECORD int-exp
Is the number of the block where you want to write the record. Since disks are direct access devices, there is no next record concept. Therefore, you must always specify the RECORD clause with the PUT statement.

If your record size is not 512 bytes, you waste space. Pad the unused space with FILL statements to make sure that no unexpected data is written to disk.

You also can write records that are longer than a block in length. For example, if your record is five blocks long (a total of 2560 bytes) and you use a RECORD 1% clause, this record is written to the first through fifth disk blocks. If you use a RECORD 2% clause to write a second record of the same size, the record is physically written to the sixth through tenth disk blocks.

```
10      ON ERROR GOTO 19000
        !
        DECLARE WORD CT
        !
        MAP (GAME) LONG REC, STRING FILL = 508%
        !
        OPEN 'DK0:' AS FILE #9%, ORGANIZATION VIRTUAL, MAP GAME
        !
        ! Use the counter index to specify RECORD value
        !
        FOR CT = 1% TO 15%
            INPUT REC
            MOVE TO #9%, REC
            PUT #9%, RECORD CT
        NEXT CT
        !
        GOTO 32766
        !
19000   ! Error Handler
        !
        PRINT 'Unexpected error: ';ERT$(ERR)
        RESUME 32766
        !
32766   CLOSE #9%
32767   END
```

In this example:

1. The OPEN statement opens a channel to the disk on DK0:. There is no RECORDSIZE clause, so the record size defaults to 512 bytes.
2. The FOR...NEXT loop writes 15 records to disk.
3. The RECORD clause uses the value of each iteration of the loop to specify which block the record is written to. Therefore, during the first loop the PUT statement writes the first record to the first block, during the second loop the PUT statement writes the second record to the second block, and so on up to block 15.

17.3.3.4 Adding New Records to Disk

To add new records to a disk, simply write them to an available block using the PUT statement with the RECORD clause. You must know which disk blocks are available to prevent writing over existing records. Note that BASIC does not warn you if you write over an existing record.

If you wrote four 128-byte records (a total of 512 bytes) to each of 30 blocks, you should be able to add records to the thirty-first block:

```
PUT #4%, RECORD 31%
```

17.3.3.5 Reading Records on Disk

You can read records from disk using the GET statement. Each GET operation retrieves an integral number of 512-byte blocks. The format of the GET statement is as follows:

GET [#]chnl-exp ,RECORD int-exp

chnl-exp
Is the number of the channel on which the disk is open. It can be preceded by an optional number sign (#).

RECORD int-exp
Is the number of the block where the record is stored. Since disks are direct access devices, there is no next record concept. Therefore, you must always specify the RECORD clause with the PUT statement.

If you define the size of each record to be 128 bytes, you can store four records in each disk block. One GET operation makes four records available for processing. For example:

```
10      ON ERROR GOTO 19000
        !
        MAP (VIR) STRING A = 128,    &
                B = 128,        &
                C = 100,        &
                FILL = 28,      &
                D = 128
        !
        OPEN 'DK0:' AS FILE #4%, ORGANIZATION VIRTUAL, MAP VIR
        !
        GET #4%, RECORD 6%
        PRINT A, B, C, D
        !
        GOTO 32766
        !
19000   ! Error Handler
        !
        PRINT 'Unexpected error: ';ERT$(ERR)
        RESUME 32766
        !
32766   CLOSE #4%
32767   END
```

In this example, the GET statement retrieves the first 512 bytes located in the sixth block. Note that the FILL statement in the MAP declaration pads 28 bytes of unused space in the third record.

Because each GET operation transfers an integral number of 512-byte disk blocks, your program must perform record blocking and deblocking. Use the MAP, MAP DYNAMIC, REMAP, or MOVE statements to perform record blocking and deblocking.

MAP statements allocate static memory. REMAP statements can redefine the memory allocated with MAP statements. For example:

```
10     MAP (VIR) STRING TOTAL_RECORD = 512%
       !
       OPEN 'DK0:' AS FILE #1%, ORGANIZATION VIRTUAL, MAP VIR
       !
       MAP DYNAMIC (VIR) STRING CURRENT_RECORD
                 !
                 FOR I% = 0% TO 3%
                 REMAP (VIR) STRING FILL = I% * 128%, CURRENT_RECORD
                 PRINT CURRENT_RECORD
                 NEXT I%
```

In this example:

1. Line 10 declares the storage and data type for 512 bytes.
2. The OPEN statement opens the disk on channel 1 and references the MAP declared in line 10.
3. The MAP DYNAMIC statement declares the data type and name of CURRENT_RECORD. At each loop iteration, the position of this variable in the buffer is redefined with the REMAP statement.

The following example uses MOVE statements to block and deblock records:

```
10     ON ERROR GOTO 19000
       !
       OPEN 'DK0:' AS FILE #3%, ORGANIZATION VIRTUAL, RECORDSIZE 512%
       !
       DECLARE WORD CT, STRING REC
       !
             FOR CT = 1% TO 15%
                 GET #3%, RECORD CT
                 MOVE FROM #3%, REC
                 PRINT REC
             NEXT CT
       !
       GOTO 32766
       !
19000  ! Error Handler
       !
       PRINT 'Unexpected error: ';ERT$(ERR)
       RESUME 32766
       !
32766  CLOSE #3%
32767  END
```

In this example:

1. The OPEN statement opens the disk on channel #3. The record size and record buffer size is 512 bytes.
2. The FOR...NEXT loop retrieves one 512-byte record from blocks 1 through 15.
3. The MOVE statement moves each record from the record buffer and assigns it to the string variable RECORD.
4. The PRINT statement displays the value of RECORD.

17.3.3.6 Locating Records on Disk

There is no statement you can use to find a record on disk. To locate records, you must know the contents of each block and then read the records in the block you specify with the RECORD clause added to the GET statement.

17.3.3.7 Deleting Records on Disk

There is no BASIC statement you can use to delete or truncate records on disk. To delete a record, you must write over it.

17.3.3.8 Dismounting a Disk

After you finish using a disk, you must follow these steps:

- If you used the MOUNT command, specify the DISMOUNT command.
- Specify the DCL command DEALLOCATE.
- Spin down and remove your disk.

Specify the physical or logical name of the disk drive you use in the DISMOUNT and DEALLOCATE commands. Consult your system manager to learn how to spin down and remove your disk.

Note

Do not remove a disk until the message "Dismount complete" is displayed on your terminal. If you do not wait for this message before removing the disk, you may corrupt data on the disk.

17.4 Network I/O

If your system supports DECnet facilities, and your computer is one of the nodes in a DECnet network, you can communicate with other nodes in the network with BASIC–PLUS–2 program statements. BASIC–PLUS–2 lets you do the following:

- Read and write files on a remote node as you do files on your own system (remote file access)
- Exchange data with a process executing at a remote location (task-to-task communication)

To write or read files at a remote site, include the node name as part of the file specification. For example:

```
OPEN "WESTON::DU1:[HOLT]TEST.DAT" FOR INPUT AS FILE #2%
```

You can also assign a logical name to the file specification, and use that logical name in all file I/O.

If the account at the remote site requires a username and password, include this *access string* in the file specification. You do this by enclosing the access string in quotation marks and placing it between the node name and the double colon. For example, the following file specification accesses the account [HOLT] on node WESTON by giving the username HOLT and the password PASWRD. After accessing the file, your BASIC–PLUS–2 program can read and write records as if the file were in your account.

```
OPEN 'WESTON"HOLT PASWRD"::DU0:[HOLT]INDEXU.DAT' &
        FOR INPUT AS FILE #1%, INDEXED, PRIMARY TEXT$
```

BASIC–PLUS–2 also supports task-to-task communication. See the *DECnet-RSX Programmer's Reference Manual* or *DECnet/E Network Programming in BASIC-PLUS and BASIC–PLUS–2* for more information on network I/O and task-to-task communication.

18 Libraries

This chapter describes how to use BASIC–PLUS–2 libraries, user-created libraries, and RMS–11 libraries to optimize program development.

18.1 Introduction

Libraries are files that can contain object modules, text modules, and executable code. If you have routines that are used in many programs, placing the routines in a library lets you access them without including the routines in the source code, thus shortening task build time and conserving disk space. After you select a library, specify the BASIC–PLUS–2 BUILD command so that the Task Builder will use the new library to task build your program.

There are three different types of libraries: libraries supplied by BASIC–PLUS–2, user-created libraries, and RMS–11 libraries. These libraries are described in the following sections.

18.2 BASIC–PLUS–2 Libraries

BASIC–PLUS–2 supplies two types of libraries: memory-resident and object module. When your program is linked, the Task Builder automatically links your program to a memory-resident library, if one is available. If a memory-resident library is not available, the Task Builder links your program to an object module library. The following sections describe the BASIC–PLUS–2 memory-resident and object module libraries.

18.2.1 BASIC–PLUS–2 Memory-Resident Libraries

A memory-resident library resides in memory and contains executable code; that is, code in which all symbols are already resolved. If you use resident libraries, the Task Builder only needs to supply an address pointing to the executable code for that routine within the resident library. Furthermore, any program using the same routine can share its executable code.

Using memory-resident libraries instead of object module libraries does not significantly affect the execution speed of a single BASIC–PLUS–2 program, but it does provide the following advantages:

- Your programs link faster because there are fewer accesses to the object module library on disk.
- Even though resident libraries require a large amount of physical memory, there is less total physical memory used for systems running many BASIC–PLUS–2 tasks.

If you do not use resident libraries, the Task Builder resolves the symbols for a given routine in the object module library.

BASIC–PLUS–2 supplies the following memory-resident libraries:

- BP2RES
- BP2SML

BP2RES contains most of the BASIC–PLUS–2 Object Time System (OTS) routines. It uses 18K words of physical memory and 8K words of virtual address space. BP2SML contains a subset of the most commonly used BASIC–PLUS–2 routines. It uses 8K words of physical memory and 8K words of virtual address space. Your system manager selects one of these libraries as the default memory-resident library during the BASIC–PLUS–2 installation.

Note

The BASIC–PLUS–2 memory-resident libraries are an installation option. You can determine if the memory-resident libraries are installed on your system by using the DCL command SHOW LIBRARY on RSTS/E systems or the DCL command SHOW COMMON on RSX systems.

Your program is linked to the default memory-resident library unless you select an alternative memory-resident library by using either the LIBRARY or BRLRES command in the BASIC–PLUS–2 environment. See Section 18.3.2 for more information on selecting a memory-resident library.

Before selecting a memory-resident library, you should first consider the memory restrictions of your system and the speed with which you want your programs to compile and link. For example, your program links faster if you use BP2RES rather than BP2SML. BP2RES contains almost all of the BASIC–PLUS–2 OTS routines, which means the Task Builder does not need to

access the BASIC–PLUS–2 object module library on disk as often. Note that you also have the option of selecting a user-created memory resident library. User-created memory-resident libraries are described in Section 18.3.

18.2.2 BASIC–PLUS–2 Object Module Libraries

An object module library resides on disk and contains object code. The object code for each routine is resolved when the program is linked and the resulting executable code is incorporated in the task image. Thus, a task image created by linking to an object module library is larger than the same program linked with a resident library.

The advantages of using an object module library are as follows:

- There is less total physical memory used for systems running few BASIC–PLUS–2 tasks.
- On small systems, using object module libraries is the only means of using BASIC–PLUS–2 and RMS–11 at the same time.

BASIC–PLUS–2 provides you with one object module library, BP2OTS.OLB. BP2OTS.OLB contains all of the BASIC–PLUS–2 OTS routines. If your system does not have memory-resident libraries, the Task Builder extracts all the BASIC–PLUS–2 routines it needs from BP2OTS.OLB. The BASIC–PLUS–2 memory-resident libraries contain most, but not all of the BASIC–PLUS–2 OTS routines. Therefore, the BASIC–PLUS–2 object module library, BP2OTS.OLB, must always be available to link BASIC–PLUS–2 programs. See the *BASIC–PLUS–2 Reference Manual* for a list of the BASIC–PLUS–2 OTS routines.

18.3 User-Created Libraries

Creating your own library can be a convenient and efficient way to access the code your program needs. You can create a memory-resident or an object module library for your program. The type of library you choose to create depends on the following:

- What library routines your programs need
- How often your programs use the code you place in a library
- The amount of virtual address space available for your tasks

You can identify the library routines a program requires by generating a map file when you build or link the program. To generate a map file, specify the /MAP qualifier with either the BASIC–PLUS–2 BUILD command or the DCL command LINK.

Once you identify those routines you use frequently, you can create your own library to contain those routines. Because user-created libraries are smaller, programs that use them link faster.

Note

DIGITAL does not support user-created libraries.

The following sections describe how to create memory-resident and object module libraries.

18.3.1 Creating a Memory-Resident Library

You can create your own memory-resident library by modifying an existing memory-resident library and then installing it. First, determine which routines your programs use most frequently. Then, modify an existing memory-resident library to contain only those routines.

Once you create a memory-resident library, you can link your program to that library. The following section tells you how to select a memory-resident library.

See the *RSTS/E Task Builder Reference Manual* or the *RSX–11M/M–PLUS Task Builder Reference Manual* for more information on creating memory-resident libraries.

18.3.2 Selecting a Memory-Resident Library

The BRLRES command allows you to specify a memory-resident library to be used when you task-build a program. When you use the BUILD command, BASIC–PLUS–2 includes the specified library in the Task Builder command file.

The format of the BRLRES command is as follows:

BRLRES [lib-param]

lib-param
Is either the file specification of a memory-resident library or the keyword NONE. The memory-resident library can be supplied by BASIC–PLUS–2 or user-created. The keyword NONE tells the Task Builder not to link your task to the default memory-resident library; therefore, the Task Builder links your task to the BASIC–PLUS–2 object module library, BP2OTS.OLB. If you do not supply a *lib-param*, BASIC–PLUS–2 prompts you for one.

The following is an example of the BRLRES command:

```
BRLRES LB:[1,1]BASIC2
```

You can also select a memory-resident library by adding the /BRLRES qualifier to the BUILD command. The library you specify remains in effect for only one BUILD operation.

See the *BASIC–PLUS–2 Reference Manual* for more information on the BRLRES command.

18.3.3 Creating an Object Module Library

You can decrease the time it takes to link your program by condensing the routines in the BASIC–PLUS–2 default object module library or by placing frequently used program code into an object module library instead of a subprogram.

After you determine what program code to place in a object module library, compile the code to produce object module files. Then, use the DCL command LIBRARY with the /CREATE qualifier to create the library and place the object files in the library. For example:

```
$ LIBRARY/CREATE MYLIB SUB1,SUB2,SUB3
```

This LIBRARY command creates the object module library MYLIB.OLB, and places the object modules SUB1.OBJ, SUB2.OBJ, and SUB3.OBJ, in that library.

The following section describes how to select an object module library to link to your program.

See the *RSTS/E System User's Guide* or the *RSX–11M–PLUS Command Language Manual* for more information on the LIBRARY command and creating object module libraries.

18.3.4 Selecting an Object Module Library

You have the following options when selecting an object module library for your program:

- You can use the default BASIC–PLUS–2 object module library.
- You can use both the BASIC–PLUS–2 object module library and your own object module library to link the program.
- You can use a customized version of the BASIC–PLUS–2 object module library or a library you create yourself.

You cannot use your object module library to replace the BASIC–PLUS–2 object module library unless your object module library includes all of the BASIC–PLUS–2 OTS routines.

18.3.4.1 Selecting Both the Default and a User-Created Object Module Library

If you want to use both the default BASIC–PLUS–2 object module library and your own object module library to link the program, you must edit the ODL file for your program to include references to your library; otherwise, the Task Builder can only link your program to the default object module library.

The following example is an ODL file before modification:

```
        .ROOT BASIC2-RMSROT-USER,RMSALL
USER:   .FCTR SY:T-LIBR
LIBR:   .FCTR LB:[1,1]BP2OTS/LB
@LB:[1,1]BASIC2
@LB:[1,1]RMS11X
        .END
```

The following example is the previous ODL file which has been modified to include references to the user-created library, MYLIB.OLB. MYLIB.OLB tells the Task Builder to link to three object module libraries: BP2OTS.OLB, SUBLIB.OLB, and RMSLIB.OLB.

```
        .ROOT BASIC2-RMSROT-USER,RMSALL
USER:   .FCTR SY:T-LIBR-MYLIB
LIBR:   .FCTR LB:[1,1]BP2OTS/LB
MYLIB:  .FCTR SY:[30,3]SUBLIB/LB
@LB:[1,1]BASIC2
@LB:[1,1]RMS11X
        .END
```

18.3.4.2 Selecting a User-Created Object Module Library

The DSKLIB command lets you select a user-created object module library to link to your program. The object module library you specify is included in the Task Builder command (CMD) file. The Task Builder searches this library when it links your program. Note that you must use the DSKLIB command before you use the BUILD command to generate CMD and ODL files.

The format of the DSKLIB command is as follows:

DSKLIB [file-spec]

file-spec
Is the file specification of a disk-resident object module library. The specified library can either be the default BASIC–PLUS–2 library or a user-created library. If you specify the DSKLIB command without a *file-spec*, BASIC–PLUS–2 prompts for one and displays the name of the current default disk-resident library. If you press the RETURN key without specifying a library file specification, the current default disk-resident library is used.

The following is an example of the DSKLIB command:

```
DSKLIB LB:[1,1]MYLIB
```

You can also select an object module library by adding the /DSKLIB qualifier to the BUILD command. The library you specify remains in effect for only one BUILD operation.

For more information on the DSKLIB command, see the *BASIC–PLUS–2 Reference Manual.*

18.4 RMS–11 Libraries

An RMS–11 library supplies RMS–11 code for file and record operations. BASIC–PLUS–2 uses RMS–11 if you specify a file organization in an OPEN statement, or if you specify a BUILD command with the /VIRTUAL, /SEQUENTIAL, /INDEXED, or /RELATIVE qualifiers. RMS–11 supplies one object module library and two memory-resident libraries. These libraries are described in the following sections.

18.4.1 The RMS–11 Memory-Resident Libraries

RMS–11 supplies the following memory-resident libraries:

- RMSRES
- DAPRES

RMSRES contains most of the routines in the RMS–11 OTS and supports sequential, relative, and indexed file organizations. DAPRES contains routines that support remote file access to other DECnet nodes. However, DAPRES cannot be used by itself. It must be used with RMSRES.

RMSRES uses 23K words of memory and 8K words of virtual address space. Like BASIC–PLUS–2, even if the RMS–11 resident library is available on your system, the RMS–11 object module library must be available to link BASIC–PLUS–2 programs that use RMS–11.

Note

The RMS–11 memory-resident libraries are an installation option. Your system manager selects the default RMS–11 resident library when installing BASIC–PLUS–2. You can determine if the RMS–11 memory-resident libraries are installed on your system by using the DCL command SHOW LIBRARY on RSTS/E systems or the DCL command SHOW COMMON on RSX systems.

If an RMS–11 memory-resident library is available, you also have the option of selecting an RMS–11 ODL file to describe how the memory-resident library will be overlaid in memory. The ODL files supplied by RMS–11 are described in Section 18.4.4.

The following section describes how to select an RMS–11 memory-resident library to link to your program.

18.4.2 Selecting an RMS–11 Memory-Resident Library

Before you select an RMS–11 memory-resident library, you should first consider the memory restrictions of your system and the type of file organization you want to use. Once you decide which RMS–11 memory-resident library you want to use, you can specify the BASIC–PLUS–2 RMSRES command. The RMSRES command determines which RMS–11 memory-resident library the Task Builder links to your program.

The format of the RMSRES command is as follows:

RMSRES lib-param

lib-param
Is either the file specification of an RMS–11 memory-resident library or the keyword NONE. The memory-resident library can either be supplied by RMS–11 or user-created. NONE tells the Task Builder not to link your task to the default RMS–11 resident library; therefore, the Task Builder links your task to the RMS–11 object module library, RMSLIB.OLB. If you do not supply a *lib-param*, BASIC–PLUS–2 prompts for one and displays the name of the current default RMS–11 library.

The following is an example of the RMSRES command:

```
RMSRES LB:[1,1]RMSRES
```

To override the default RMS–11 resident library, use the RMSRES command before you use the BUILD command. The RMS–11 resident library you specify in the RMSRES command remains in effect until you either specify a new RMSRES command or exit from the BASIC–PLUS–2 environment. Once you exit from the BASIC–PLUS–2 environment, the default RMS–11 resident library is used.

You can also select an RMS–11 memory-resident library by using the /RMSRES qualifier with the BUILD command. The library you specify remains in effect for only one BUILD operation.

18.4.3 The RMS–11 Object Module Library

RMS–11 provides you with one object module library, RMSLIB.OLB. RMSLIB.OLB contains all RMS–11 Object Time System (OTS) routines. Even if both RMS–11 memory-resident libraries are available, the RMS–11 object module library must always be available, since some routines only exist in this library.

To determine if RMSLIB.OLB is available on your system, enter the ODL file for your program on your terminal. You should see a line that looks like the following:

```
@LB:[1,1]name
```

Here, *name* represents the file name of an ODL file that accesses the RMS–11 object module library, RMSLIB.OLB. You can use one of the ODL files RMS–11 supplies to access the RMS–11 object module library. The RMS–11 ODL files are described in the following section.

18.4.4 RMS–11 ODL Files

RMS–11 Overlay Description Language (ODL) files tell the Task Builder how to overlay segments of the RMS–11 object module library and memory-resident libraries. Your system manager selects the default RMS–11 ODL file during installation. You can determine the default RMS–11 ODL file on your system by using the SHOW command in the BASIC–PLUS–2 environment.

RMS–11 supplies you with seven different ODL files: two ODL files for the RMS–11 memory-resident libraries and five ODL files for the RMS–11 object module library. The RMS–11 ODL files are described in Table 18–1.

Table 18–1 ODL Files Supplied by RMS–11

ODL File	Description
ODL Files for the RMS–11 Memory-Resident Libraries	
RMSRLX	Requires 8K words of virtual address space and includes support for the RMSRES memory-resident library.
DAPRLX	Requires 8K words of virtual address space, clusters the DAPRES and RMSRES memory-resident libraries, and allows you to use DECnet to access remote nodes.

(continued on next page)

Table 18–1 (Cont.) ODL Files Supplied by RMS–11

ODL File	Description
ODL Files for the RMS–11 Object Module Library	
RMS11S	Requires 7K bytes of virtual address space, overlays RMSLIB.OLB in 11 segments, and supports sequential and relative file organizations.
RMS12S	Requires 9K bytes of virtual address space, overlays RMSLIB.OLB in 5 segments, and supports sequential and relative file organizations.
RMS11X	Requires 10K bytes of virtual address space, overlays RMSLIB.OLB in 35 segments, and supports sequential, relative, and indexed file organizations.
RMS12X	Requires 12K bytes of virtual address space, overlays RMSLIB.OLB in 13 segments, and supports sequential, relative, and indexed file organizations.
DAP11X	Requires 14K bytes of virtual address space, overlays RMSLIB.OLB in 16 segments, and supports sequential, relative, and indexed file organizations. It also supports file access to other computer systems through the use of DECnet, if DECnet is installed on your system.

Note

ODL file names may change with new versions of RMS. Therefore, refer to the RMS–11 distribution kit for current ODL file names.

The following section tells you how to select an RMS–11 ODL file.

18.4.5 Selecting an RMS–11 ODL File

Before selecting an ODL file, you should consider the following:

- The type of file organization you want to use
- The amount of available virtual address space
- The time in which you want your program to execute

If you want to use an indexed file organization, for example, but have limited address space, you would choose RMS11X.ODL to describe the overlay structure of RMSLIB.OLB. Because RMS11X.ODL overlays RMSLIB.OLB in 35 segments, your task takes longer to execute than if you were to use RMS12X.ODL, which overlays RMSLIB.OLB in 13 segments.

To select an ODL file, use the BASIC–PLUS–2 ODLRMS command. The format of the ODLRMS command is as follows:

ODLRMS odl-param

odl-param
Is either the file specification of an RMS–11 ODL file or the keyword NONE. The ODL file can either be supplied by RMS–11 or user-created. NONE specifies no ODL file.

The following is an example of the ODLRMS command:

```
ODLRMS LB:[1,1]RMSRLX.ODL
```

Specify the ODLRMS command before you use the BUILD command. The Task Builder includes the ODL file you specify in the Task Builder command (CMD) file until you either specify a new ODL file the ODLRMS command or exit from the BASIC–PLUS–2 environment. Once you exit from the BASIC–PLUS–2 environment, the default ODL file is used.

You can also select an RMS–11 ODL file by using the the /ODLRMS qualifier with the BUILD command. The ODL file you specify remains in effect for only one BUILD operation.

See the *RSTS/E RMS–11 User's Guide* or the *RSX–11M/M–PLUS RMS–11 User's Guide* for information on creating ODL files.

18.5 Clustering Memory-Resident Libraries

When you link your task to two or more memory-resident libraries, the libraries use additional virtual memory space. Therefore, the more memory-resident libraries you use, the less space is available for your task. You can avoid this problem by clustering your memory-resident libraries. When you cluster memory-resident libraries, the libraries share Active Page Registers (APRs) which reduces the amount of virtual memory space required and consequently increases the space available for your task.

To cluster memory-resident libraries, use the /CLUSTER qualifier with the BASIC–PLUS–2 BUILD or SET commands. For example:

```
BUILD /CLUSTER=DAPRES
```

When you specify the /CLUSTER qualifier, BASIC–PLUS–2 adds a line to the Task Builder CMD file telling the Task Builder to cluster the following libraries:

- The default BASIC–PLUS–2 memory-resident library; you must link a BASIC–PLUS–2 memory-resident library to your task for the /CLUSTER qualifier to have any effect

- The default RMS–11 memory-resident library (if required for your program)
- The library you specify with the /CLUSTER qualifier; for example, if you have DECnet installed on your system, you may want to include support for file access to other computer nodes by specifying the DAPRES RMS–11 resident library

In addition to the default BASIC–PLUS–2 resident library, there must be at least one other resident library you are linking to your task for the /CLUSTER qualifier to have an effect.

18.6 Remote File Access

If DECnet is provided on your system, your BASIC–PLUS–2 programs can access files on other computer systems. To open a file on another computer node, you include the node name in the file specification. For example:

```
OPEN 'boston::db1:[30,42]file.dat' AS FILE #1, SEQUENTIAL
```

This OPEN statement specifies the device DB1: and the account [30,42]. When this statement executes, RMS–11 searches for the file FILE.DAT, on a computer system called BOSTON. You must specify a device and account name or RMS–11 searches for the file in the default DECnet account on a remote node.

There are two ways to include remote file access support for your BASIC–PLUS–2 programs:

- With the object module library, use the RMS–11 ODL file DAP11X to include DECnet support in the RMSLIB object module library.
- With memory-resident libraries, cluster the RMSRES and DAPRES memory-resident libraries with the BASIC–PLUS–2 memory-resident library.

To cluster the RMS–11 memory-resident libraries RMSRES and DAPRES with a BASIC–PLUS–2 memory-resident library, you specify RMSRES as the RMS–11 memory-resident library and DAPRES as the CLUSTER library. For example:

```
RMSRES RMSRES

BASIC2

SET /CLUSTER : DAPRES

BASIC2
```

When you use the RMS–11 memory-resident library RMSRES with another memory-resident library, you must use RMSRLX as the RMS–11 ODL file. However, RMSRES and DAPRES are both RMS–11 libraries and must both be overlaid. Therefore, if you cluster RMSRES and DAPRES, you must use a special RMS–11 ODL file, DAPRLX. To specify DAPRLX, use the /ODLRMS qualifier or ODLRMS command. For example:

```
ODLRMS LB:DAPRLX

BASIC2
```

See Chapter 12 and Chapter 17 for more information on accessing files on remote nodes.

19 Utilities

This chapter describes three BASIC–PLUS–2 utilities: the Optimizer Utility, the Dump Analyzer Utility, and the Resequencer Utility.

19.1 The Optimizer Utility

The BASIC–PLUS–2 Optimizer Utility reduces the size of an object module by replacing duplicate calls to the BASIC–PLUS–2 Object Time System (OTS) with a subroutine.

The Optimizer Utility is primarily useful for executing large tasks or when the amount of available memory is low. When using the Optimizer Utility, follow these guidelines for maximum performance:

- Use your fastest disk. The Optimizer utility creates a large workfile in your default directory and uses it extensively. Fast access to this workfile is a major factor in its performance.
- Submit your optimization job as a batch job, when system load is minimal.
- Specify the largest segment size possible.
- Generate a listing file only when you want to determine the proper segment size.

19.1.1 Invoking the Optimizer Utility

You can either use the OPT command or the DCL command RUN $BP2OPT to invoke the Optimizer Utility. These commands are described in the following sections. The Optimizer Utility is an installation option. See your system manager if you have trouble invoking the Optimizer Utility.

19.1.1.1 The OPT Command

The OPT command must be installed as a CCL or MCR command on your system for you to use it to invoke the Optimizer Utility.

The format of the OPT command is as follows:

OPT [/qualifier . . .] file-spec [/qualifier . . .]

/qualifier
Is the name of a qualifier that indicates a specific action to be performed by the Optimizer Utility on the specified file.

file-spec
Is the file specification of the object module to be optimized. If you do not supply a file type, the Optimizer Utility searches for a file with a file type of OBJ by default.

Qualifier	Default
/[NO]LIST [= file-spec]	/NOLIST
/[NO]OUTPUT = file-spec	/NOOUTPUT
/SEGMENT_SIZE = int-const	/SEGMENT_SIZE=3

Command Qualifiers

/[NO]LIST [=file-spec]
The /LIST qualifier causes the Optimizer to generate a listing file. The listing file contains a MACRO representation of the code being optimized and also contains optimization statistics. The optimization statistics can help you determine the correct segment size to specify for optimizing your task. If you do not specify a file specification with the /LIST qualifier, the name of the listing file is the same as the name of the input file with a file type of LST. The default is /NOLIST.

/[NO]OUTPUT = file-spec
The /OUTPUT qualifier allows you to specify a file specification for the generated object module. If you do not specify a *file-spec*, or specify /NOOUTPUT, the default file specification for the object file is the file name of the program and a file type of OBJ.

/SEGMENT_SIZE = int-const
The /SEGMENT_SIZE qualifier specifies the minimum word size of the program segments to be optimized. *Int-const* can be an integer value from 3 through 32767. The default is /SEGMENT_SIZE = 3. See Section 19.1.2 for more information on choosing a segment size.

The following is an example of the OPT command:

```
$ OPT BIGTSK.OBJ/SEGMENT_SIZE=25/LIST=OPTLST.LIS
```

This command optimizes all program segments containing more than 25 words in the object module BIGTSK, and generates the Optimizer listing file, OPTLIST.LIS.

19.1.1.2 The RUN $BP2OPT Command

The DCL RUN $BP2OPT command invokes the Optimizer Utility interactively and causes it to prompt you for information about your task. The format of the RUN $BP2OPT command is as follows:

RUN $BP2OPT

When you invoke the Optimizer Utility with the RUN $BP2OPT command, the Optimizer Utility displays a line identifying itself and then prompts you for the following information:

- The name of the input file
- The name of the output file
- The name of the listing file you want to generate, if any
- The segment size you want optimized

The following is an example of the RUN $BP2OPT command and the Optimizer Utility dialogue:

```
$ RUN $BP2OPT
BASIC-PLUS-2 Optimizer V1.0
Input File  ? TEST.OBJ
Output File ? TEST.OBJ
Listing File? TEST.LST
Minimum Segment Size? 18
```

The following defaults apply:

- You must supply the name of an input file; there is no default.
- The default ouput file is the same file name and file type as the input file.
- If you press Return in response to the Listing file? prompt, no listing file is created.
- The segment size determines the size of the program segments to be optimized. The segment size can be an integer value from 3 through 32767. The default segment size is 3.

The following section describes how to select the segment size that is right for optimizing your program.

19.1.2 Choosing a Segment Size

The segment size you specify determines the word size of the program segments that will be optimized. You specify a segment size with either the /SEGMENT_SIZE qualifier to the OPT command, or as a response to the Optimizer dialogue when you invoke the Optimizer interactively. You can specify a segment size of 3 through 32767. A segment size of 3 (the default) causes the Optimizer to replace all duplicate program segments of more than three words with a subroutine.

The best way to choose a segment size for your program is to experiment with choosing several different segment size values and determining which value provides the most optimization. You can determine how much optimization is caused by a particular segment size by generating an Optimizer listing file.

The Optimizer listing file provides the following information:

- A MACRO representation of the initial code
- A MACRO representation of the code after optimization
- An Optimizer Statistics section. This section provides the following information:
  - The name of the input file
  - The name of the output file
  - The specified segment size
  - The initial size of the object module
  - The final size of the object module after using the Optimizer
  - The percentage of memory saved by using the Optimizer

If you do not request a listing file, the Optimizer Statistics section of the listing is displayed on your terminal by default when the Optimizer finishes executing.

After generating an Optimizer listing, you can determine whether the segment size you specified was adequate for the object module, by looking at the size of the object module after optimization and the percentage of memory saved.

In general, a larger segment size provides the least optimization. A small segment size provides the most optimization because it optimizes small segments as well as large, but it also slows execution time because it increases the amount of branching in the program to and from the generated subroutines. Therefore, if task execution time is of primary importance, it is better to specify a large segment size rather than a small one.

Example 19–1 is an example of an Optimizer listing file for the following program:

```
10   FIND <pound-sign>Chnl, KEY #Key_one EQ "Jones"
20   GET  #Chnl, KEY #Key_one GT "Smith"
```

This listing was generated from the following command line:

```
$ OPT/LIST TESTOPT.OBJ
```

Example 19–1 Example of an Optimizer Listing File

```
**************     BASIC-PLUS-2 Threaded Code Optimizer V1.0     **************

**************                  ❶Original Code                      **************
************** $CODE PSECT of BASIC-PLUS-2 Main Program 'TESTOPT'**************
     0                   L9                  ; Label operand
     1                   TKB supplied        ; Program limits
     2                    0                  ; Constant operand
     3                   $$BP2  + 0          ; PSECT+offset operand
     4                   $FLAGR + 0          ; PSECT+offset operand
     5                   Expression          ; Complex Relocation
     6                    0                  ; Constant operand
     7                   $ICIO1 + 0          ; PSECT+offset operand
     8                    0                  ; Constant operand
     9 L9:               L22                 ; Label operand
    10                   $PDATA + 0          ; PSECT+offset operand
    11                   $PDATA + 0          ; PSECT+offset operand
    12                   $IDATA + 0          ; PSECT+offset operand
    13                    4                  ; Constant operand
    14                   $STRNG + 0          ; PSECT+offset operand
    15                    0                  ; Constant operand
    16                   $TDATA + 0          ; PSECT+offset operand
    17                   $ARRAY + 0          ; PSECT+offset operand
    18                    0                  ; Constant operand
    19                    20559              ; Constant operand
    20                    13140              ; Constant operand
    21                    8224               ; Constant operand
    22 L22:    LIN$                          ; Labeled, Branch, Thread
    23                    10                 ; Constant operand
```

(continued on next page)

Example 19–1 (Cont.) Example of an Optimizer Listing File

```
❷  24         MOF$MS                              ; Thread
   25                      $IDATA + 4            ; PSECT+offset operand
   26        CIF$                                ; Thread
   27        MOF$MS                              ; Thread
   28                      $IDATA + 0            ; PSECT+offset operand
   29        CIF$                                ; Thread
   30        RLI$M                               ; Thread
   31                      $PDATA + 10           ; PSECT+offset operand
   32        LFK$                                ; Thread
   33                       0                    ; Constant operand
   34        LIN$                                ; Branch, Thread
   35                       20                   ; Constant operand
❸  36         MOF$MS                              ; Thread
   37                      $IDATA + 4            ; PSECT+offset operand
   38        CIF$                                ; Thread
   39        MOF$MS                              ; Thread
   40                      $IDATA + 0            ; PSECT+offset operand
   41        CIF$                                ; Thread
   42        RLI$M                               ; Thread
   43                      $PDATA + 0            ; PSECT+offset operand
   44        LGK$                                ; Thread
   45                       2                    ; Constant operand
   46        FLN$                                ; Branch, Thread
   47                       20                   ; Constant operand
❹  48         END$                                ; Thread
**************              End of $CODE PSECT                **************
```

(continued on next page)

Example 19–1 (Cont.) Example of an Optimizer Listing File

```
**************              ❺Results of Optimization                **************
************** $CODE PSECT of BASIC-PLUS-2 Main Program 'TESTOPT'**************
    0                        L9                     ; Label operand
    1                        TKB supplied           ; Program limits
    2                         0                     ; Constant operand
    3                        $$BP2  + 0             ; PSECT+offset operand
    4                        $FLAGR + 0             ; PSECT+offset operand
    5                        Expression             ; Complex Relocation
    6                         0                     ; Constant operand
    7                        $ICIO1 + 0             ; PSECT+offset operand
    8                         0                     ; Constant operand
    9 L9:                    L22                    ; Label operand
   10                        $PDATA + 0             ; PSECT+offset operand
   11                        $PDATA + 0             ; PSECT+offset operand
   12                        $IDATA + 0             ; PSECT+offset operand
   13                         4                     ; Constant operand
   14                        $STRNG + 0             ; PSECT+offset operand
   15                         0                     ; Constant operand
   16                        $TDATA + 0             ; PSECT+offset operand
   17                        $ARRAY + 0             ; PSECT+offset operand
   18                         0                     ; Constant operand
   19                         20559                 ; Constant operand
   20                         13140                 ; Constant operand
   21                         8224                  ; Constant operand
   22 L22:    LIN$                                  ; Labeled, Branch, Thread
   23                         10                    ; Constant operand
❻   24         GSU$                                  ; Thread
   25                        L41                    ; Label operand
   26         RLI$M                                 ; Thread
   27                        $PDATA + 10            ; PSECT+offset operand
   28         LFK$                                  ; Thread
   29                         0                     ; Constant operand
   30         LIN$                                  ; Branch, Thread
   31                         20                    ; Constant operand
```

(continued on next page)

Example 19–1 (Cont.) Example of an Optimizer Listing File

```
❼ 32          GSU$                                 ; Thread
  33                       L41                     ; Label operand
  34         RLI$M                                 ; Thread
  35                       $PDATA + 0              ; PSECT+offset operand
  36         LGK$                                  ; Thread
  37                        2                      ; Constant operand
  38         FLN$                                  ; Branch, Thread
  39                        20                     ; Constant operand
 ❽40          END$                                 ; Thread
 ❾41 L41:     MOF$MS                               ; Labeled, Thread
  42                       $IDATA + 4              ; PSECT+offset operand
  43         CIF$                                  ; Thread
  44         MOF$MS                                ; Thread
  45                       $IDATA + 0              ; PSECT+offset operand
  46         CIF$                                  ; Thread
  47         REG$                                  ; Branch, Thread
**************              End of $CODE PSECT              **************

**************             ❿Optimizer Statistics             **************
Input File : TESTOPT
List File  : TESTOPT
Minimum segment size: 3
$CODE words initial : 49
$CODE words final   : 48
$CODE percent saved : 2.04082
**************                End of Listing                **************
```

❶ This is the **Original Code** section. It contains the MACRO representation of the object module before optimization.

❷ Line 24 through line 29 is the first segment of duplicate code. Its size is 6 words.

❸ Line 36 through line 41 is the second segment of duplicate code.

❹ Line 48 is the END$ thread. It marks the end of a main program. If this was a subprogram, the thread SBE$ would appear here instead of END$.

❺ This is the **Results of Optimization** section. It contains the MACRO representation of the object module after optimization.

❻ Line 24 is the GSU$ thread. It replaces lines 24 through 29 in the original code and causes execution to branch to line 41.

❼ Line 32 is the second GSU$ thread. It replaces lines 36 through 41 in the original code and causes execution to branch to line 41.

❽ Listing line 40 is the END$ thread that marks the end of the module.

9. Entry point L41 is the start of the GSU$ subroutine created by the Optimizer. Subroutines created by the Optimizer always start after an END$ or SBE$ thread.

10. This is the **Optimization Statistics** section. It provides the following information:

    - TESTOPT is the name of the input file.
    - TESTOPT is the name of the output file.
    - The specified segment size is 3.
    - The size of the object module before optimization was 49 words.
    - The size of the object module after optimization was 48 words.
    - The Optimizer saved 1 word of threaded code which is 2.0% of the space the threaded code used to occupy.

    Note that the final object module size does not reflect the amount of memory that program data (such as strings and and variables) will require.

If the specified segment size was 6 for this program, the optimization labeled L41 would not be made because it is only 6 words in length. Thus, the segment size limits the number of optimization made, and in so doing, minimizes the run-time effect of using the Optimizer Utility.

It is recommended that you only generate a listing file when determining a segment size, as it slows optimization time.

19.1.3 Optimizer Error Messages

The following is an alphabetical list of the Optimizer Utility error messages and the user action required to correct them.

Corrupted or Non-BP2 object module

Explanation: The input file cannot be optimized because it is either not a BASIC–PLUS–2 generated object module, or it is corrupt. You cannot use non-BASIC object modules with the Optimizer Utility.

User Action: If the input file is a BASIC–PLUS–2 generated object module, try recompiling the program. If this does not correct the error, submit an SPR.

Duplicate qualifier, value superseded

Explanation: You specified duplicate qualifiers on the same command line. The value of the qualifier was superseded by the rightmost occurrence.

User Action: None.

Error opening listing file: <name>

Explanation: The listing file could not be opened.

User Action: Take action based on second line of the error message.

Error opening input file: <name>

Explanation: The Optimizer Utility could not open the input file.

User Action: Take action based on the second line of the error message.

Error opening output file: <name>

Explanation: The Optimizer Utility could not open the output file.

User Action: Take action based on the second line of the error message.

Error opening work file: <name>

Explanation: The work file could not be opened.

User Action: Take action based on the second line of the error message.

Error renaming output to: <name1>, output left in file: <name2>

Explanation: The Optimizer Utility optimized the input file but could not generate an output file with the name you specified. The optimized object module is left in the file <name2>.

User Action: Use the DCL command RENAME to rename the object module.

Extraneous input ignored

Explanation: Extra characters were encountered at the end of the command line. The Optimizer ignored the extraneous characters.

User Action: None.

File name expected

Explanation: You neglected to specify a file specification when you invoked the Optimizer Utility.

User Action: Re-invoke the Optimizer Utility and specify a file specfication.

Internal error

Explanation: This error should never occur.

User Action: Submit an SPR.

Illegal qualifier

Explanation: You specified an invalid Optimizer qualifier.

User Action: Either specify a valid qualifier or remove the qualifier.

Illegal segment size, using default value (3)

Explanation: You specified an illegal segment size. The segment size must be in the range from 3 through 32767. The Optimizer ignores the specified segment size and optimizes your program with a segment size of 3.

User Action: None.

NO not valid for /SEGMENT_SIZE

Explanation: /NOSEGMENT_SIZE is an illegal qualifier.

User Action: Remove the qualifier.

Program contains too many <item> to be optimized

Explanation: The program contained too many of <item> to be optimized.

User Action: Divide the module into several SUB or FUNCTION subprograms and reinvoke the Optimizer Utility.

Qualifier expected

Explanation: You specified a backslash (\ ) without specifying a qualifier to the OPT command.

User Action: Either remove the backslash or specify a valid qualifier.

Qualifier value expected

Explanation: You specified a qualifier which requires an argument without specifying the argument.

User Action: Either specify an argument or remove the qualifier.

Unexpected Error in module <name>

Explanation: An unexpected error occurred during processing.

User Action: Take action based on second line of the error message.

19.2 The Dump Analyzer Utility

The BASIC–PLUS–2 Dump Analyzer Utility is only available on RSTS/E systems. Use the RSX–11M/M–PLUS Crash Dump Analyzer for similar functionality on RSX systems.

The Dump Analyzer Utility examines the binary dump generated by a program that aborts due to a FATAL error. It returns information about string space, I/O buffers, and available memory.

To invoke the Dump Analyzer, enter the following command at DCL level:

```
$ RUN $BP2DA
```

Note

If the BP2DA command has been installed as a CCL command on RSTS/E systems, or as an MCR command on RSX systems, you can simply enter BP2DA at the DCL prompt to invoke the Dump Analyzer. The Dump Analyzer Utility is an installation option. See your system manager if you are having trouble invoking the Dump Analyzer.

When you invoke the Dump Analyzer, it prompts you for the following information:

- The name of the file you want analyzed
- The name of the output file that you want the dump analysis information placed in
- Whether you want information about string space allocations included in the analysis
- Whether you want information about I/O buffer space allocations in the analysis
- Whether you want an analysis done of the whole program
- Whether you want information about low core memory in the analysis; if you enter Yes in response to this prompt, the output file includes the contents of the registers, stack status, and I/O buffers

For example:

```
Input File Name <SY:PMDnnn.PMD> : ?
Output File Name <SY:PMDnnn.DPA> : ?
String space <Yes> ?
I/O Buffers <Yes> ?
Entire Program <Yes> ?
Low Core <No> ?
```

In the Dump Analyzer prompts, all defaults are displayed in angle brackets (<>). The BP2DA program automatically assigns a file name in the format PMDnnn.PMD for the input file, where nnn is the current job number. If you do not supply the name of an output file, the dump analysis information is written to a file with the same name as the input file and a file type of DPA.

If the dump file is too large for available memory, the Analyzer returns the error "?Maximum memory exceeded." You must then either re-run the program and select fewer options for analysis, or use the PMDUMP program. See the *RSTS/E System User's Guide*for more information on the PMDUMP program.

19.3 The Resequencer Utility

The BASIC–PLUS–2 Resequencer Utility renumbers program lines (and references to those lines) throughout a specified program. With the Resequencer, you can divide a program into a maximum of 10 segments and specify a different resequencing scheme for each segment. The Resequencer can only be used on programs containing a maximum of 2500 line numbers.

Enter the following RUN command to invoke the Resequencer Utility from DCL:

```
$ RUN $B2RESQ
```

Note

If the B2R command has been installed as a CCL command on RSTS/E systems or as an MCR command on RSX systems, you can simply enter B2R at the DCL prompt to invoke the Resequencer Utility. The Resequencer Utility is an installation option. See your system manager if you are having trouble invoking the Resequencer.

When you invoke the Resequencer, it prompts you for the following information:

- The name of the input file
- The name of the output file

- The number of subprograms in the input file to be resequenced
- The beginning line number of the first subprogram to be resequenced
- The last line number of the subprogram to be resequenced
- The new line number for the first line in the subprogram
- The increment at which you want each line resequenced

For example:

```
Enter BASIC-PLUS-2 program to resequence? MAINPROG.B2S
Enter output file?  RESEQ_MAINPROG.B2S
Number of program segments to be resequenced? 2
Segment n old beginning line number? 300
Old end line number? 900
New beginning line number? 1100
New increment this segment? 15
```

The following defaults apply:

- If no file type is specified for the input or output files, the Resequencer assumes a file type of B2S.
- If you do not specify the number of program segments to be resequenced, the Resequencer resequences the entire program starting at line 10 and incrementing by 10.
- If you do not specify a line number in the original program where you want resequencing to begin, resequencing begins at the first line number in the program.
- If you do not specify a line number in the original program where you want resequencing to end, resequencing ends at line number 32767 or the last line in the input program.
- If you do not specify a new line number for the first line of the resequenced segment, line number 10 is assigned to the first resequenced line.
- If you do not specify a resequencing increment, all line numbers are resequenced at an increment of 10.

After the last prompt, the Resequence Utility renumbers the program according to the segment definitions. It then updates line number references in control statements (for example, GOTO and THEN) to reflect the new sequential order. If you specified more than one subprogram to be resequenced, the Resequencer then displays the last four prompts again, until all specified subprograms are resequenced.

19.3.1 Creating a Resequencer Command File

Instead of responding to the Resequencer prompts interactively, you can include your answers in an indirect command file by using a text editor. Then, you invoke the Resequencer, and supply the name of the indirect command file at the prompt "Number of program segments to be resequenced?". For example:

```
Name of input file? MAINPROG.B2S
Name of output file? RESEQ_MAINPROG.B2S
Number of program segments to be resequenced? @RESEQ.CMD
```

Here, RESEQ.CMD is the name of an indirect command file containing Resequencer commands. The Resequencer Utility then renumbers the program MAINPROG.B2S according to the commands contained in the indirect command file.

Note that if you do not supply a file type for the indirect command file, the Resequencer assumes a file type of CMD by default.

The following section describes how to format Resequencer commands in an indirect command file.

19.3.2 Formatting Commands in a Resequencer Command File

The format for Resequencer commands in an indirect command file is as follows:

command:[,command[: . . . command]]

command — Is one of three resequence commands summarized in Table 19–1. You can continue commands on the next line with the ampersand (&) continuation character.

colon (:) — Is a command separator. You end each command except the last with a colon.

comma (,) — Is a segment separator. A segment is a unique group of program lines.

Table Table 19–1 lists and describes the Resequencer commands.

Table 19–1 Resequencer Commands

Command	Meaning
O m-n	Resequence the segment defined as line numbers m through n. The default for m is 1; the default for n is 32767.
N m	Begin resequencing the segment at line m. The default for m is line 10.
I d	Increment line numbers by a value of d. The default is 10.

The following is an example of Resequencer commands in an indirect command file:

```
O1-100:N10:I1,O150-200:N50,  &
O1000-10000:N1000:I50
```

These commands cause the following resequencing changes to occur:

Command	Change
O1-100:N10:I1	Resequences old line numbers 1 through 100 (O1-100:), starting at line 10 (N10:) and incrementing by 1 (I1).
O150-200:N50	Resequences old line numbers 150 through 200 (O150-200:), starting at line 50 (N50). Because the command line specified no increment value, the default is 10.
O1000-10000:N1000:I50	Resequences old line numbers 1000 through 10000 (O1000-10000:), starting at line 1000 (N:1000) and incrementing by 50 (I50).

19.3.3 Resequencer Utility Error Messages

The Resequence Utility prints an error message when it detects a resequencing error. The following is a list of the Resequencer Utility error messages and an explanation of each error.

?Input file not found

Explanation: The Resequence Utility cannot find your input file.

?Input line numbers <line numbers> are out of strictly ascending order

Explanation: The line numbers in the file are not in ascending order. You can reorder program lines by doing the following:

- Invoking the BASIC–PLUS–2 compiler
- Bringing the program into memory with the OLD command
- Issuing the REPLACE command to save the program

The compiler reorders the line numbers and returns the corrected program to your directory. You can then resequence the corrected program.

?Invalid segment parameters

Explanation: You specified an end line number lower than the starting line number.

%Line # not found, resequencing continuing

Explanation: A program line does not have the required line number.

?Maximum of ten segments allowed

Explanation: The Resequencer command line specifies more than 10 segments.

?Output name must be different from file name

Explanation: The output and input file names must be different.

?Proposed resequencing out of integer bounds

Explanation: The line numbers of a segment will exceed 32767 when resequenced.

?Proposed resequencing overlaps

Explanation: The new program segments will overlap each other when resequenced.

?Resequenced segment encompasses unresequenced line

Explanation: A segment being resequenced will overlap a line you did not specify for resequencing.

?Segment descriptors overlap

Explanation: You entered a line number in more than one segment.

?Specification file error - expecting more command data

Explanation: Your command file is not correctly formatted.

?Specification file not found

Explanation: The Resequence Utility cannot find your indirect command file.

?Syntax error, number too large for integer

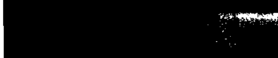
Explanation: The command line contained an integer outside the valid range (1%-32767%).

?Syntax error

Explanation: The command line input contained an error.

?Two segments have identical new beginning value

Explanation: Two program segments cannot start with the same line number.



20
Optimization Techniques

This chapter is for the experienced programmer. It provides you with information that can help you write transportable programs and describes techniques that you can use to improve task execution time.

20.1 Writing Transportable Programs

This section describes some programming practices that can help you write BASIC programs that are transportable between PDP–11 operating systems and VMS systems. Transportable code should be modular and carefully designed to isolate incompatibilities between different operating systems. Incompatibilities are usually the result of using a feature that is available on only one operating system.

Isolate references to system-specific features by placing them in subprograms or external functions, or by conditionally executing them using the %IF-%THEN-%ELSE-%END-%IF directive. This helps to minimize changes to your program and prevents the entire program logic from being tied to these features.

You can also place code that calls system specific features in object module libraries. See Chapter 18 for information on creating object module libraries.

Use the statements and functions in Table 20–1 to avoid using system-specific features that perform the same function.

Table 20–1 BASIC–PLUS–2 Substitutes for System Services

Statement or Function	Effect
CTRLC function	Passes control to an error handler when a Ctrl/C is typed
RCTRLC function	Disables the effect of CTRLC

(continued on next page)

Table 20–1 (Cont.) BASIC–PLUS–2 Substitutes for System Services

Statement or Function	Effect
RCTRLO function	Cancels the effect of Ctrl/O and resumes display
ECHO function	Displays a character entered on the terminal
NOECHO function	Does not display characters as entered
ONECHR function	Allows single-character input
NAME...AS statement	Changes a file name
KILL statement	Deletes a file from disk
SLEEP statement	Suspends program execution

Do not use the FSS$ function. This function evaluates file specifications that differ from system to system and therefore are not transportable. If you must use the FSS$ function to evaluate a file specification, place it in a subprogram.

Avoid chaining when possible. Instead of chaining to the code you want, place that code in subprograms. Data can be shared between program modules by opening files and by using COMMON and MAP statements.

20.2 Optimizing Your Program

Optimizing your program means improving the performance of your program, decreasing the time it takes to link it, and increasing the amount of virtual address space for your task. To optimize a program, identify the routines that take the longest to execute and that you use the most often. You can then use the techniques described in the following sections to optimize those routines.

20.2.1 I/O Operations

If your program performs file I/O with sequential or block file organization, use MAP statements instead of MOVE statements to decrease the time it takes to execute your program.

MAP statements use static storage, execute faster, and give you direct access to named variables. You can also use multiple MAP statements to access that storage in different ways.

MOVE statements use dynamic storage. The record size is determined at run time, and each MOVE statement physically moves data. Although MOVE statements offer flexibility, programs that contain them execute slower than programs using MAP statements.

Dynamic mapping (MAP, MAP DYNAMIC, and REMAP statements) is by far the best way to handle dynamic data. Dynamic mapping provides the performance of MAP statements, the flexibility of FIELD and MOVE statements, and supports all data types.

Also, use LINPUT statements instead of INPUT LINE statements. The INPUT LINE statement includes all line terminators whereas the LINPUT statement truncates them.

20.2.2 Assigning Variables

When assigning variables, use the MACRO routine described in Chapter 11 to initialize variables in COMMON and MAP statements instead of using READ and DATA statements. COMMON and MAP statements use no virtual address space and do not affect run time.

20.2.3 Choosing Compiler Options

Use the /NOLINE qualifier to the COMPILE or SET commands to generate smaller object modules and faster execution times. The /LINE qualifier to the COMPILE command keeps track of which program line is executing. When you use the /NOLINE qualifier, BASIC–PLUS–2 does not have to update the line your program is executing. If you use the /NOLINE qualifier, you cannot use the ERL function or the RESUME statement without a target to handle errors.

20.2.4 Selecting Data Types

Use the following guidelines when choosing data types for program variables:

- The BASIC–PLUS–2 Object Time System (OTS) routines can process operations using integer numbers faster than floating-point numbers; therefore, use integers whenever possible.
- If you are given a choice, use WORD integers instead of BYTE integers because WORD integers can be processed faster and offer a larger range of values. Use BYTE integers instead of LONG integers because the OTS routines require separate arithmetic routines to process LONG integers.
- Use integer numbers for the following program variables:
  - Loops
  - Channel expressions
  - Counters
  - Flags
  - Array indexes

- If you do not have a Floating-Point Processor (FPU) and you are performing arithmetic operations, use LONG integers instead of floating-point numbers or strings. BASIC–PLUS–2 run-time code requires approximatley 250 instructions to emulate floating-point arithmetic, whereas LONG integers require fewer instructions.
- The OTS takes longer to process arithmetic operations using DOUBLE floating-point numbers; therefore, use SINGLE floating-point numbers.
- Avoid data conversions such as mixing integers and floating-point numbers in expressions. Data conversion requires additional code that slows program execution. For example, if you want a faster program execution time, you would use the program code shown in example 1 rather than the program code in example 2.

  Example 1
  ```
  X% = Y% * 2%
  ```
  Example 2
  ```
  X% = Y% * 2
  ```

20.2.5 Arithmetic Operations

Use the following guidelines when performing arithmetic operations:

- Avoid using string arithmetic operations. String arithmetic requires much more OTS code to be brought into your task, which slows execution.
- If you must do string arithmetic, avoid division. Division takes longer to process than any other type of arithmetic operation. Use multiplication instead. For example, the code in the first example executes faster than the code in the second.

  Example 1
  ```
  A$ = PROD$ ( B$, ".5", X% )
  ```
  Example 2
  ```
  A$ = QUO$ ( B$, "2", X% )
  ```

20.2.6 Using Control Structures

Use the following guidelines for optimizing your use of control structures:

- IF statements generate less code than logical expressions and, consequently, execute faster. The code shown in example 1, for instance, executes faster than the code in example 2:

  Example 1
  ```
  A% = 1% IF Y% = 2 IF X% > 5%
  ```

Example 2

```
IF ( X% > 5% ) AND ( Y% = 2% ) THEN A% = 1%
```

- ON GOTO and ON GOSUB statements generate less code and execute faster than IF and SELECT statements. However, IF and SELECT statements improve the readability and, consequently, the maintainability of programs.

  SELECT statements offer the added advantage of conditionally executing several blocks of code, each dependent upon a separate conditional expression. IF statements, on the other hand, choose between only two blocks of code. To conditionally execute more than two blocks of code with IF statements, you must nest them. For example, use the program construct shown in example 1 rather than the program constructs in examples 2 and 3.

Example 1

```
10     ON PART_CODE% GOTO Enter, Update, Delete
   .
   .
   .
   Enter:
   .
   .
   .
     GOTO Choice
   Up_date:
   .
   .
   .
     GOTO Choice
   De_lete:
   .
   .
   .
     GOTO Choice
```

Example 2

```
10     IF ITEM% = 1%
          THEN ...
          ELSE IF ITEM% = 2%
               THEN ...
               ELSE IF ITEM% = 3%
                    THEN ...
                    ELSE ...
```

Example 3

```
10      SELECT ITEM%
              CASE = 1%
    .
    .
    .
              CASE = 2%
    .
    .
    .
              CASE = 3%
    .
    .
    .
```

- Using GOSUB statements instead of DEF statements generates smaller tasks and faster task execution; however, there is no local error handling for GOSUB routines. Use the program construct in example 1 rather than the one shown in example 2.

Example 1

```
10      B = X
        C = Y
        GOSUB 2000
        PRINT A
    .
    .
    .
2000    A = B + C
        RETURN
```

Example 2

```
10      DEF INTEGER ADD(B,C) = B + C
20      PRINT ADD(X,Y)
```

- Using DEF statements instead of SUB and FUNCTION subprograms generates less code; therefore, your program executes faster. However, DEF statements cannot overlay virtual address space. To optimize your task execution time, use the program code shown in example 1 rather than the code shown in examples 2 and 3.

Example 1

```
10      DEF REAL ADD(X,Y)
           ADD = X + Y
        END DEF
    .
    .
    .
        PRINT 'The sum is  '; DEF ADD(A,B)
```

Example 2

```
10      EXTERNAL SUB ADD (REAL,REAL,REAL)
        CALL ADD(A,B,C)
        PRINT 'The sum is  '; C

300     SUB ADD(REAL A,REAL B,REAL RESULT)
        RESULT = A + B
        END SUB
```

Example 3

```
10     EXTERNAL REAL FUNCTION ADD(REAL,REAL)
   .
   .
   .
       PRINT 'The sum is ';ADD(A,B)

300    FUNCTION REAL ADD(REAL A,REAL B)
            ADD = A + B
       END FUNCTION
```

20.2.7 Selecting Libraries

Use memory-resident libraries to improve task-build time and memory usage when you run more than two executable images at the same time. The Task Builder can resolve program code much faster using a resident library than using an object module library. Using the large RMS resident library (RMSRES), in particular, substantially improves the Task Builder's performance; however, the task you link to RMSRES may not execute as quickly as it would normally.

You can also limit the diversity of BASIC–PLUS–2 code you use in your programs, and then customize your resident libraries to include only those routines your programs need. Note that Digital does not support customized libraries. See Chapter 18 for information on libraries.

20.2.8 RMS–11 Operations

Use the following guidelines for optimizing RMS–11 operations:

- When possible, use a SEQUENTIAL file organization instead of an INDEXED file organization. The code to support indexed files requires more virtual address space and consequently executes slower.
- If your programs do not require RMS–11, use the /NOSEQ, /NOREL, /NOVIR qualifiers to the BUILD command or use NONE as the argument to the RMSRES and ODLRMS commands to prevent the Task Builder from linking to unnecessary code.

- If possible, close files in the reverse order that you open them. For example, if you have three files and you know that there is one file that needs to be open only for a short time, open that file last. If you open and close files in this sequence, it frees I/O space for future use.
- Use a larger bucketsize value to bring in more records at once, reducing the number of accesses to the file on disk.

See the *RSX-11M/M-PLUS RMS-11 User's Guide* or the *RSTS/E RMS-11 User's Guide* for information on RMS-11 optimization techniques.

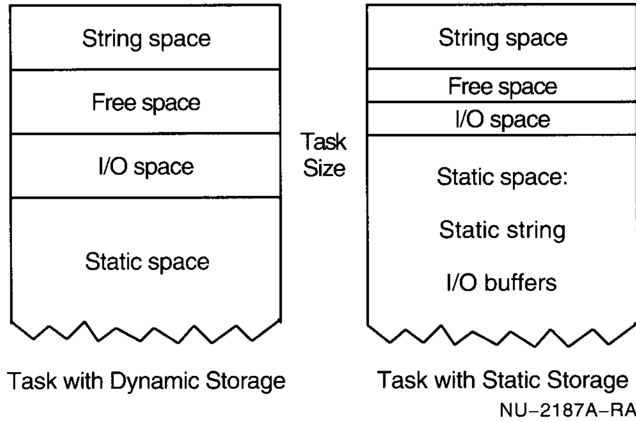
20.2.9 Static and Dynamic Storage

BASIC-PLUS-2 dynamic memory is made up of string, free, and I/O space. String space is used for all dynamic strings, I/O space is used for all file handling, and free space shrinks and expands as string and I/O space is used.

Statically allocating memory with MAP and COMMON statements generally speeds program execution but wastes space if the allocation is not entirely used. Although dynamically allocating memory generally slows program execution, it does not waste space and offers greater flexibility than statically allocating memory.

Figure 20-1 compares the size of a task without static storage to the size of a task with static storage. This figure demonstrates that a task with static storage increases the amount of available memory.

Figure 20–1 Comparison of Static Storage to Dynamic Storage



20.2.10 Extending Memory

Although BASIC–PLUS–2 automatically pre-extends memory when a task is loaded, this extension may not be sufficient. Generally, dynamic string handling and file I/O require more dynamic space, whereas record buffering requires more static space.

If you think your task needs more memory, it is more efficient to pre-extend memory allocation with the EXTTSK option added to the Task Builder command file. The maximum extension is limited by the size of your task. See the *RSX–11M/M–PLUS Task Builder Reference Manual* or the *RSTS/E Task Builder Reference Manual* for more information.

A
Compile-Time and Environment Error Messages

This appendix lists the BASIC–PLUS–2 compile-time and environment errors, their causes, and the user action required to correct them.

A.1 Diagnosing Compile-Time and Environment Errors

BASIC–PLUS–2 signals a compile-time error if an error occurs during compilation. BASIC–PLUS–2 signals an environment error if you enter an illegal command or attempt an illegal operation while in the BASIC–PLUS–2 environment. For compile-time errors, BASIC–PLUS–2 also does the following:

- Indicates the program line that generated the error or errors
- Displays the program line
- Shows you the location of the error or errors and assigns a number to each location for future reference
- Displays the statement number within the line, the location number as previously displayed, and the error message text; this is repeated for each error in the line

BASIC–PLUS–2 repeats this procedure for each error diagnosed during compilation.

There are four different levels of compile-time and environment errors: information, warning, error, and fatal. These errors are described as follows:

- Information Level Errors

  Information errors do not cause the compilation to abort, they simply provide information.
- Warning Level Errors

  Warning level errors do not cause the compilation to abort; however, they do indicate that some unusual or uncommon events have occured.

- Error Level Errors

  Error level errors do not cause the compilation to abort. When an error level error occurs, the remainder of the program is checked for additional errors; however, no object module file is generated when an error level error occurs.

- Fatal Level Errors

  Fatal level errors cause the compilation to abort. The remainder of the program is not checked for additional errors. No object module is generated.

A.2 Error Message Format

The format for compile-time and environment error messages is as follows:

? S <n>, <n>: <message>

S <n>	Is the nth statement within the displayed line (the statement containing the error)
<n>:	Is the nth error within the line's "picture"
<message>	Is the text of the error message

For example:

```
Error on line 10

        10 DECLARE REAL BYTE A, A
........................1.......2

? S 1, 1:   conflicting data type specifications
? S 1, 2:   illegal multiple definition of name A
```

A.3 Alphabetical List of Error Messages

This section contains a list of all BASIC–PLUS–2 compile-time and environment error messages. Most of these error messages are alphabetized by the first word in the error message as it appears on your terminal screen; however, error messages that begin with a number or name unique to the current operation are alphabetized according to the next word in the error message.

$ only allowed with STRING keyword

Explanation: ERROR—The dollar sign ($) suffix is only allowed on STRING variables.

User Action: Remove the dollar sign ($) from the variable name or change the data type of the variable.

% only allowed with BYTE, WORD, LONG, or INTEGER keywords

Explanation: ERROR—The percent sign (%) suffix is only allowed on INTEGER variables.

User Action: Remove the percent sign (%) suffix or change the data type of the variable to INTEGER.

actual argument must be specified

Explanation: ERROR—A DEF function reference contains a null argument, for example, FNA(1,,2).

User Action: Specify all arguments when referencing a DEF function.

an internal coding error has been detected. Submit an SPR

Explanation: ERROR—An internal error occurred during the compilation.

User Action: Submit an SPR and include all relevant information.

array not allowed in DEF declaration

Explanation: ERROR—The parameter list for a DEF function definition contained an entire array.

User Action: Remove the array specification; you cannot pass an entire array as a parameter to a DEF function.

array <name> too large

Explanation: ERROR—The program contains a MAP or COMMON area of storage longer than 32767 bytes.

User Action: Reduce the length of the COMMON or MAP storage.

attempt to sequence over existing statement

Explanation: WARNING—A SEQUENCE command specifies a starting line number that already exists in the BASIC–PLUS–2 source program in memory.

User Action: Specify a starting line number higher than any existing line or delete the old statement before using the SEQUENCE command.

attributes of overlaid variable <name> don't match

Explanation: WARNING—A variable name appears in more than one overlaid MAP; however, the attributes specified for the variable are inconsistent.

User Action: If the same variable name appears in multiple overlaid MAPs, the attributes (for example, data type) must be identical.

attributes of prior reference to <name> don't match

Explanation: WARNING—A variable or array is referenced before the MAP that declares it. The attributes of the referenced variable do not match those of the declaration.

User Action: Make sure that the variable or array has the same attributes in both the reference and the declaration.

bad line number pair

Explanation: ERROR—A compiler command specifies nonexistent line numbers or a pair of line numbers out of sequence, for example, LIST 100—2.

User Action: Specify only existing line numbers and specify the numbers in correct numeric order when specifying a range.

BASIC—'@' must be at beginning of filename

Explanation: ERROR—An at sign (@) was not the first character in the file specification of an indirect command file.

User Action: Precede the file specification with an at sign.

BASIC—ambiguous qualifier, supply additional characters

Explanation: ERROR—You abbreviated a qualifier, but did not specify enough characters.

User Action: Add more characters to the qualifier.

BASIC—argument contradicts previous setting

Explanation: ERROR—You specified an argument to a qualifier that conflicts with another argument on the command line.

User Action: Remove the argument.

BASIC—argument expected

Explanation: ERROR—You followed a qualifier with the equals sign (=) or colon (:) separator, but did not specify an argument.

User Action: Either specify an argument or delete the separator character.

BASIC—argument must be numeric

Explanation: ERROR—You specified a non-numeric argument with a qualifier that only accepts numeric arguments.

User Action: Specify a numeric argument to the qualifier.

BASIC—argument not allowed

Explanation: ERROR—You specified an argument with a qualifier that does not accept arguments.

User Action: Remove the argument.

BASIC—argument not in range <number> to <number>

Explanation: ERROR—You specified a numeric argument that is not in the allowed range.

User Action: Specify a number within the allowed range.

BASIC—argument required

Explanation: ERROR—You failed to specify an argument with a qualifier that requires an argument.

User Action: Specify an argument.

BASIC—/BUILD not valid in this context

Explanation: WARNING—The /BUILD qualifier is not valid with the command you specified. You may only use the /BUILD qualifier when compiling a program at DCL level with the DCL command BASIC.

User Action: Remove the /BUILD qualifier.

BASIC—invalid qualifier

Explanation: ERROR—You specified an illegal qualifier with the BASIC command.

User Action: Remove the qualifier.

BASIC—invalid argument

Explanation: ERROR—You specified an illegal argument with a qualifier.

User Action: Specify a legal argument.

BASIC—list file specification not allowed in this context

Explanation: WARNING—You can only specify a file specification with the /LIST qualifier when compiling a program at DCL level with the DCL command BASIC.

User Action: Remove the file specification.

BASIC—macro file specification not allowed in this context

Explanation: WARNING—You can only specify a file specification with the /MACRO qualifier when compiling a program at DCL level with the DCL command BASIC.

User Action: Remove the file specification.

BASIC—NO prefix not allowed

Explanation: ERROR—You specified a NO prefix with an argument to the qualifier. A NO prefix is not allowed with the argument.

User Action: Remove the NO prefix.

BASIC—object file specification not allowed in this context

Explanation: WARNING—You can only specify a file specification with the /OBJECT qualifier when compiling a program at DCL level with the DCL command BASIC.

User Action: Remove the file specification.

BASIC—qualifier expected

Explanation: ERROR—You specified a backslash (\ ) without the name of a qualifier.

User Action: Append the name of a qualifier to the backslash or remove the backslash.

BASIC—qualifier contradicts previous qualifier

Explanation: ERROR—You specified a qualifier that conflicts with another qualifier on the command line.

User Action: Remove the qualifier.

BASIC—wildcards not permitted - <filename>

Explanation: ERROR—You specified a wildcard in place of a file name. Using wildcards for file names is not allowed.

User Action: Specify the file name of an existing file.

bound cannot be specified for array

Explanation: ERROR—An EXTERNAL statement declaring a SUB or FUNCTION subprogram specifies bounds in an array parameter. For example:

```
EXTERNAL SUB XYZ (LONG DIM(1,2,3))
```

User Action: Remove the array parameter's bound specifications. When declaring an external subprogram you can specify only the number of dimensions for an array parameter. For example:

```
EXTERNAL SUB XYZ (LONG DIM(,,))
```

bounds must be specified for array

Explanation: ERROR—The program contains an array declaration that does not specify the bounds (maximum subscript value). For example:

```
DECLARE LONG A(,)
```

User Action: Supply bounds for the declared array. For example:

```
DECLARE LONG A(50,50)
```

BUFFER inconsistent with MODE

Explanation: ERROR—The BUFFER and MODE clauses cannot be contained in the same OPEN statement.

User Action: Remove either the BUFFER or MODE clause.

built in function not supported

Explanation: ERROR—The program contains a reference to a built-in function not supported by this version of BASIC–PLUS–2.

User Action: Remove the function reference.

built in function requires numeric expression

Explanation: ERROR—The program specifies a string expression for a built-in function that requires a numeric argument.

User Action: Supply a numeric expression for the built-in function.

cannot change /DEBUG after LOAD command

Explanation: ERROR—An attempt was made to change the /[NO]DEBUG qualifier after an object module was loaded.

User Action: Use the SCRATCH command to clear memory; change the /[NO]DEBUG qualifier to the desired default and then recompile and reload the object module so that the loaded object module corresponds with the present environment defaults for /DEBUG.

can't continue

Explanation: ERROR—A CONTINUE command was typed after changes had been made to the source code.

User Action: After changes have been made to the source code, you can only run the program; you cannot continue it.

can't find psect <name>

Explanation: FATAL—An internal error occurred when the BASIC–PLUS–2 LOAD or RUN command was executed, indicating that the compiler could not find a MAP or compiler PSECT.

User Action: Submit an SPR including all relevant information.

CHAIN does not support line-number clause

Explanation: ERROR—A CHAIN statement contains a LINE keyword and a line-number argument.

User Action: Remove the LINE keyword and the line-number argument.

CHANGE statement is ambiguous

Explanation: ERROR—A string variable and a numeric array have the same name in a CHANGE statement.

User Action: Change the name of the string variable or the numeric array.

CHANGES not allowed on primary key

Explanation: ERROR—The PRIMARY KEY clause in an OPEN statement specifies CHANGES.

User Action: Remove the CHANGES keyword; you cannot change the value of a primary key.

CHANGES requires DUPLICATES

Explanation: WARNING—RMS–11 does not support indexed files with the CHANGES and NODUPLICATES for ALTERNATE KEYs.

User Action: Add DUPLICATES to the ALTERNATE KEY clause of the OPEN statement.

channel expression must be numeric

Explanation: ERROR—The program contains a non-numeric channel expression, for example, PUT #A$.

User Action: Change the channel expression to be numeric.

COMMON/MAP <name> is too large

Explanation: ERROR—The program contains a MAP or COMMON area of storage longer than 32767 bytes.

User Action: Reduce the length of the COMMON or MAP storage.

conflicting data type specifications

Explanation: ERROR—The program contains a declarative statement containing two or more consecutive and contradictory data type keywords, for example, DECLARE REAL BYTE.

User Action: Remove one of the data type keywords or make sure that the keywords refer to the same generic data type, for example, DECLARE REAL SINGLE is valid.

constant <name> not allowed in assignment context

Explanation: ERROR—The program tries to assign a value to a user-defined constant.

User Action: Remove the assignment statement; once you have assigned a value to a declared constant, you cannot change it.

constant expression required

Explanation: ERROR—A statement specifies a variable, built-in function reference or exponentiation where a constant is required.

User Action: Supply an expression containing only literals or declared constants, or remove the exponentiation operation.

constant is inconsistent with the type of <name>

Explanation: ERROR—A DECLARE CONSTANT statement specifies a value that is inconsistent with the data type of the constant, for example, a BYTE value specified for a REAL constant.

User Action: Change the declaration so that the data type of the value matches that of the constant.

current scale factor is <number>

Explanation: WARNING—A SCALE command did not specify a scale factor. The SCALE command is ignored and the present scale factor displayed.

User Action: None.

data type keyword not allowed in SUB statement

Explanation: ERROR—A SUB statement contains a data type keyword between the subprogram name and the parameter list.

User Action: Remove the data type keyword. In a SUB statement, data type keywords can appear only within the parameter list.

data type required for variable <vbl-name> with /EXPLICIT

Explanation: ERROR—A program compiled with the /TYPE:EXPLICIT qualifier declares a variable without specifying a data type.

User Action: Supply a data type keyword for the variable or compile the program without the /TYPE:EXPLICIT qualifier.

data type required in EXTERNAL CONSTANT declaration

Explanation: ERROR—An EXTERNAL CONSTANT statement has no data-type keyword.

User Action: Supply a data-type keyword to specify the data type of the external constant.

DEF <name> mode not as declared

Explanation: ERROR—The specified data type in a function declaration disagrees with the data type specified in the function definition.

User Action: Make the data type specifications match in both the function declaration and the function definition.

DEF <name> not defined

Explanation: ERROR—The program contains a reference to a non-existent user-defined function.

User Action: Define the function in a DEF statement.

DEF invocation not allowed in assignment context

Explanation: ERROR—A DEF function invocation (including a parameter list) appears on the left side of an assignment statement.

User Action: Remove the assignment statement. You cannot assign values to a function invocation.

DEF* formal <formal-name> inconsistent with usage outside DEF*

Explanation: ERROR—A DEF* formal parameter has the same name as a program variable but has different attributes.

User Action: You should not use the same names for DEF* parameters or program variables. If you do, you must make sure that they have the same data type and size.

directive must be only item on line

Explanation: ERROR—The program contains a compiler directive that is not the only item on the line.

User Action: Place the directive on its own line.

directive not valid in immediate mode

Explanation: ERROR—A compiler directive was typed in the BASIC–PLUS–2 environment.

User Action: None. Compiler directives are invalid in immediate mode.

division by zero

Explanation: WARNING—The value of a number divided by zero is indeterminate.

User Action: Change the expression so that no expression is divided by the constant zero.

DOUBLE constant required

Explanation: ERROR—The program contains a DECLARE DOUBLE CONSTANT statement that specifies an expression for the constant value.

User Action: Remove the expression. You can specify only literal values when declaring floating-point constants.

duplicate line number <n> encountered

Explanation: WARNING—The source program has two lines with the same line number. The duplicate line replaces previous occurrence.

User Action: Add a unique line number to one of the lines.

duplicate OPEN clause

Explanation: ERROR—An OPEN statement contains more than one clause of the same type.

User Action: Remove one of the clauses.

DYNAMIC attribute only valid for MAP areas

Explanation: ERROR—A COMMON keyword is followed by the DYNAMIC keyword.

User Action: Remove the DYNAMIC keyword. The DYNAMIC attribute is valid only for MAP areas.

ELSE appears in improper context, ignored

Explanation: ERROR—The program contains an ELSE clause that either is not preceded by an IF statement or that appears after an IF has been terminated with a line number or END IF statement.

User Action: Remove either (1) the ELSE clause, (2) the terminating line number, or (3) the END IF statement.

END IF appears in improper context, ignored

Explanation: ERROR—The program contains an END IF statement that either is not preceded by an IF statement or occurs after an IF has been terminated by a line number.

User Action: Supply an IF statement or remove the terminating line number.

end of DEF seen while not in DEF

Explanation: ERROR—An FNEND or END DEF statement has no preceding DEF statement.

User Action: Define the function before inserting an END DEF statement, or delete the END DEF statement.

end of FUNCTION while not in FUNCTION

Explanation: ERROR—The program contains a FUNCTIONEND or END FUNCTION statement without an accompanying FUNCTION statement.

User Action: Supply a function subprogram or remove the FUNCTIONEND statement.

end of line does not terminate IFs due to active blocks

Explanation: ERROR—A THEN or ELSE clause contains a loop block, and a line number terminates the IF-THEN-ELSE block before the end of the loop block.

User Action: Make sure that any loop is entirely contained in the THEN or ELSE clause.

end of SUB seen while not in SUB

Explanation: ERROR—A subprogram has a SUBEND or END SUB statement without a preceding SUB statement.

User Action: Supply a SUB statement as the first statement in the subprogram or delete the END SUB or SUBEND statement.

entire array may not be passed BY VALUE

Explanation: ERROR—The program specifies BY VALUE as the passing mechanism for an entire array.

User Action: You cannot pass an entire array BY VALUE. Specify either BY REF or BY DESC.

entire array not allowed in this context

Explanation: ERROR—The program specifies an entire array in a context that permits only array elements, for example, specifying an entire array in a PRINT statement.

User Action: Remove the reference to the entire array and specify individual array elements.

entire virtual array cannot be a parameter

Explanation: ERROR—The program attempts to pass an entire virtual array as a parameter.

User Action: None. You cannot pass an entire virtual array as a parameter.

error deleting <file-name>

Explanation: ERROR—An error was detected in attempting to delete a file.

User Action: Supply a valid file specification, or take corrective action based on the associated message.

error opening compiler work files on SY:

Explanation: FATAL—The BASIC–PLUS–2 workfile could not be created.

User Action: Verify the default account is accessible and that enough disk space is available to create the workfiles.

error opening output file <file-name>

Explanation: FATAL—The COMPILE command includes an illegal output file specification, for example, COMPILE $BASIC.

User Action: Specify a valid file name.

error opening file

Explanation: ERROR—The file specified in a %INCLUDE directive could not be opened. This error message is followed by the specific RMS–11 error.

User Action: Take appropriate action based on the associated RMS–11 error.

executable DIMENSION illegal for static array

Explanation: ERROR—A DIMENSION statement names an array already declared with a DECLARE, COMMON, or MAP statement, or one that was declared statically in a previous DIMENSION statement.

User Action: Remove the executable DIMENSION statement, or originally declare the array as executable in a DIMENSION statement.

exit from DEF while not in DEF

Explanation: ERROR—An FNEXIT or EXIT DEF statement has no preceding DEF statement.

User Action: Define the function before inserting an FNEXIT or EXIT DEF statement.

exit from FUNCTION while not in FUNCTION

Explanation: ERROR—An EXIT FUNCTION or FUNCTIONEXIT statement was encountered in a module that is not a FUNCTION subprogram.

User Action: Remove the EXIT FUNCTION or FUNCTIONEXIT statement.

exit from PROGRAM while not in main program

Explanation: ERROR—A DEF, FUNCTION, or SUB subprogram contains an EXIT PROGRAM statement.

User Action: Remove the EXIT PROGRAM statement.

exit from SUB seen while not in SUB

Explanation: ERROR—A program contains an EXIT SUB or SUBEXIT statement with no preceding SUB statement.

User Action: If the program is a subprogram, supply a SUB statement; otherwise, remove the EXIT SUB or SUBEXIT statement.

expecting IF directive

Explanation: ERROR—The program contains a %END directive that is not immediately followed by a %IF directive.

User Action: Supply a %IF directive immediately following the %END directive.

expecting unary operator or legal lexical operand

Explanation: ERROR—A compiler directive contains an invalid lexical expression, for example, %IF *3% %THEN.

User Action: Correct the lexical expression.

explicit declaration of <name> required

Explanation: ERROR—The program is compiled with the /TYPE:EXPLICIT qualifier in effect and the program contains a variable, constant, function, or subprogram that is not explicitly declared.

User Action: Explicitly declare the data type of the variable, constant, function, or subprogram or compile the program without the /TYPE:EXPLICIT qualifier.

expression not allowed in this context

Explanation: ERROR—The program contains an expression in a context that allows only simple variables, array elements, or entire arrays.

User Action: Remove the expression.

expression too complicated

Explanation: The program contains an expression too complicated to compile.

User Action: Rewrite the expression as two or more less complicated expressions.

external globals not allowed—<name>

Explanation: FATAL—A program module being loaded or run with the BASIC–PLUS–2 LOAD or RUN command references an external variable or constant.

User Action: To reference external variables or constants you must link your program and run it from the system monitor level.

EXTERNAL name too long, truncating to <new-name>

Explanation: ERROR—An EXTERNAL statement names a symbol longer than six characters.

User Action: Shorten the symbol name. External names must be six characters or less.

EXTERNAL STRING variables not supported

Explanation: ERROR—The program contains an EXTERNAL statement that specifies an external string variable.

User Action: Remove or change the EXTERNAL statement. BASIC–PLUS–2 does not support external string variables.

extra ELSE directive found

Explanation: ERROR—The program contains a %ELSE directive that is not matched with a %IF directive.

User Action: Make sure that each %ELSE directive is preceded by a %IF directive and that each %IF directive contains no more than one %ELSE clause.

extra END IF directive found

Explanation: ERROR—A program unit contains a %END %IF without a preceding %IF directive.

User Action: Supply a %IF for the %END %IF.

extra left parenthesis in expression

Explanation: ERROR—A compiler directive contains a lexical expression with an extra left parenthesis.

User Action: Remove the extra parenthesis.

extra right parenthesis in expression

Explanation: ERROR—A compiler directive contains a lexical expression with an extra right parenthesis.

User Action: Remove the extra parenthesis.

failure in loading object file

Explanation: ERROR—Either an attempt was made to load a non-BASIC object module, or the compiler could not find the object file referenced by a CALL statement or EXTERNAL FUNCTION reference.

User Action: If the object file resides in the VAX/VMS Run-Time Library, you must link the program at DCL level. If the object file is in a user-supplied library, use the DCL LIBRARY command to make the missing object module available. You can load only BASIC object modules.

FIELD valid only for dynamic string variables

Explanation: ERROR—A FIELD statement contains a numeric or fixed-length string variable.

User Action: Remove the numeric or fixed-length string variable. Only dynamic string variables are valid in FIELD statements.

FIELDed variable <vbl-name> cannot be a parameter

Explanation: ERROR—The parameter list in a reference to a DEF statement or a subprogram contains a string variable or string array element that also appears in a FIELD statement.

User Action: None. If a variable appears in a FIELD statement, the variable cannot be passed as a parameter.

file access error for INCLUDE directive file <file-name>

Explanation: FATAL—The file named in the %INCLUDE directive was correctly opened but could not be read for some reason. For example, the disk drive was switched off line.

User Action: Take action based on the associated RMS–11 error messages.

FILL not allowed in DYNAMIC MAP

Explanation: ERROR—A DYNAMIC MAP statement contains a FILL item.

User Action: Remove the FILL item.

floating point error or overflow

Explanation: WARNING—The program contains a numeric expression whose value is outside the valid range for floating-point numbers.

User Action: Modify the expression so that its value is within the allowable range.

FORM FEED must appear at end of line

Explanation: WARNING—A form feed character is followed by other characters on the same line.

User Action: Remove the characters following the form feed. A form feed must be the last or only character on a line.

formal parameter must be supplied for <name>

Explanation: ERROR—The declaration of a DEF, SUB, or FUNCTION routine contains the parentheses for a parameter list, but no parameters.

User Action: Supply a parameter list or remove the parentheses.

formal string parameters may not be FIELDed

Explanation: ERROR—A variable name appears both in a subprogram formal parameter list and a FIELD statement in the subprogram.

User Action: Remove the variable from the FIELD statement or the parameter list.

found <item> when expecting <item>

Explanation: ERROR—The program contains a syntax error. BASIC–PLUS–2 displays the item at which the error was detected, then displays one or more items that make more sense in that context. The compilation continues so that other errors may be detected. The actual program line remains unchanged and no object file is produced.

User Action: Examine the line carefully to discover the error. Change the program line to correct the syntax error.

function invocation not allowed in assignment context

Explanation: ERROR—An external function invocation (including a parameter list) appears on the left side of an assignment statement.

User Action: Remove the assignment statement. You cannot assign values to a function invocation.

function nested too deep

Explanation: ERROR—The program contains too many levels of function definitions within function definitions.

User Action: Reduce the number of nested functions.

<name> has a passing mechanism specified with no parameter list

Explanation: ERROR—A CALL statement, external function reference, or EXTERNAL statement specifies a BY clause but does not specify a formal parameter list.

User Action: Remove the BY clause or supply a parameter list.

IDENT directive may appear only once per module

Explanation: WARNING—The program contains more than one %IDENT directive.

User Action: Remove all but one %IDENT directive.

IDENT directive name is too long

Explanation: WARNING—The quoted string in a %IDENT directive is too long.

User Action: Reduce the length of the string. The maximum length is six characters.

IF directive expression must be terminated by THEN

Explanation: ERROR—A %IF directive contains a %ELSE clause with no intervening %THEN clause.

User Action: Insert a %THEN clause.

IF directive in INCLUDE directive needs END IF directive in same file

Explanation: ERROR—A %INCLUDE directive that is modified by a %IF directive is missing a %END %IF directive.

User Action: Supply a %END %IF directive.

<n> IF statement(s) not terminated

Explanation: ERROR—The program contains an IF-THEN-ELSE statement within a block (for example, a FOR-NEXT, SELECT-CASE, or WHILE block), and the end of the block was reached before the IF-THEN-ELSE statement was terminated.

User Action: Check program logic to be sure IF-THEN-ELSE statements are terminated with a line number or an END IF statement before the end of the block is reached.

illegal argument for command

Explanation: ERROR—An argument was entered for a command that does not take an argument, or an invalid argument was entered for a command, for example, SCALE A or LIST A.

User Action: Re-enter the command with the proper arguments.

illegal argument passing mechanism

Explanation: ERROR—The program specifies an invalid argument passing mechanism, for example, passing strings or arrays BY VALUE, or passing an entire virtual array.

User Action: Check all elements for proper parameter passing mechanism.

illegal character <ASCII code>

Explanation: WARNING—The program contains illegal or incorrect characters.

User Action: Examine the program for correct usage of the BASIC–PLUS–2 character set; possibly delete the character.

illegal character <ASCII code> in external name

Explanation: ERROR—The external symbol in an EXTERNAL FUNCTION or CONSTANT declaration contains an invalid character.

User Action: Remove the invalid character. External names can use only RAD-50 characters.

illegal character <ASCII value> in IDENT directive

Explanation: ERROR—A %IDENT directive contains an illegal character with the reported ASCII value.

User Action: Remove the illegal character.

illegal constant type

Explanation: ERROR—The program contains an invalid declaration, for example, DECLARE RFA CONSTANT.

User Action: Remove the invalid data type. You cannot declare constants of the RFA data type.

illegal I/O channel

Explanation: ERROR—A constant channel expression is greater than 12 or less than zero.

User Action: Change the channel expression to be within the range zero to 12.

illegal library name

Explanation: ERROR—A compiler command or qualifier specifies an invalid library name, for example, DSKLIB NONE.

User Action: Specify a valid library name.

illegal line format or missing continuation at line <n>

Explanation: ERROR—A new line in the source file either does not begin with a line number, does not start with a space or tab (specifying an implied continuation), or does not follow a line ending with an ampersand (&).

User Action: Add a line number or a space or a tab to the beginning of the line, or add an ampersand to the end of the previous line.

illegal line number

Explanation: ERROR—A line number outside the valid range was typed.

User Action: Enter only line numbers in the range 1 to 32767, inclusive.

illegal line number in CHAIN

Explanation: ERROR—A CHAIN with LINE statement specifies an invalid line number. Either the number is outside the valid range, or a string expression follows the LINE keyword.

User Action: Supply an integer line number between 1 and 32767, inclusive.

illegal loop nesting, expecting NEXT <VARIABLE>

Explanation: ERROR—The program contains overlapping loops.

User Action: Examine the program logic to make sure that the FOR and NEXT statements for the inside loop lie entirely within the outside loop.

illegal matrix operation

One of the following explanations and user actions:

Explanation: ERROR—You attempted matrix division.

User Action: Remove the attempted matrix division. BASIC–PLUS–2 does not support this operation.

Explanation: ERROR—You attempted to invert a LONG or BYTE matrix.

User Action: Change the matrix to be WORD, SINGLE, or DOUBLE.

Explanation: ERROR—You attempted to invert a matrix recieved as a parameter.

User Action: Copy the matrix to a local matrix to perform the operations.

Explanation: ERROR—You attempted to transpose a matrix to itself.

User Action: Use another matrix as the destination and assign the result to the source.

Explanation: ERROR—You attempted to perform MAT multiplication with the same matrix as both the source and destination.

User Action: Use an intermediary matrix to hold the result.

illegal mode mixing

Explanation: ERROR—The program contains string and numeric operands in the same operation.

User Action: Change the expression so that it contains either string or numeric operands, but not both.

illegal multiple definition of name <name>

Explanation: ERROR—The program uses the same name for the following:

- More than one variable
- A variable and a MAP
- A variable and a COMMON
- A MAP and COMMON

User Action: Use unique names for variables, COMMONs, and MAPs.

illegal operation for argument

Explanation: ERROR—The program performs an operation that is inconsistent with the data type of the arguments, for example, an arithmetic operation on variables of the RFA data type.

User Action: Remove the operation or change the data type of the arguments.

illegal string operator

Explanation: ERROR—The program specifies an invalid string operation, for example, A$ = B$–C$.

User Action: Replace the invalid operator.

illegal switch usage—<text>

Explanation: ERROR—An invalid qualifier was specified with a compiler command, for example, SET/ABC.

User Action: Specify a valid qualifier.

illegal usage of FIELDed variable

Explanation: ERROR—Either (1) a MOVE TO or MOVE FROM statement contains a string variable or string array element that also appears in a FIELD statement, or (2) a MAT statement operates on a string array element that appears in a FIELD statement.

User Action: Either (1) remove the variable from the FIELD statement or the MOVE statement, or (2) remove the array from the MAT statement.

illegal use of unary operator

Explanation: ERROR—A compiler directive contains an invalid lexical expression, for example, %IF 1 - - 2, or %IF 1 NOT 2.

User Action: Correct the invalid lexical expression.

illegally formed name

Explanation: ERROR—The program contains an invalid user identifier (such as a variable, constant, or function name).

User Action: Change the name to comply with the rules for naming user identifiers. See the *BASIC–PLUS–2 Reference Manual* for more information.

illegally formed numeric constant

Explanation: ERROR—The program contains either (1) an invalid E-format expression or (2) a numeric constant with a digit that is invalid in the specified radix, for example, a decimal constant containing a hexadecimal digit.

User Action: Supply a valid E-format expression or a constant that is valid in the specified radix.

illegally nested DEFs

Explanation: ERROR—The program contains a DEF function block within another DEF function block.

User Action: Remove the inner DEF block. A DEF block cannot contain another DEF block.

implied continuation not allowed

Explanation: ERROR—The program contains an implied continuation line after a statement that does not allow implicit continuation, for example, a REM statement.

User Action: Use an ampersand (&) to continue the statement.

implied declaration not allowed for <name> with /TYPE=EXPLICIT

Explanation: ERROR—A program compiled with the TYPE=EXPLICIT qualifier contains an implicitly declared variable.

User Action: Compile the program without the TYPE=EXPLICIT qualifier, or declare the variable explicitly.

inaccessible code follows line <n> statement <m>

Explanation: WARNING—The program contains one or more statements that cannot be accessed, for example, a multi-statement line whose first statement is GOTO, EXIT, ITERATE, RESUME, or RETURN.

User Action: Make sure that the GOTO, EXIT, ITERATE, RESUME, or RETURN statement is the only statement on a numbered line or the last statement on a multi-statement line.

INCLUDE directive file must be on a random access device

Explanation: ERROR—A %INCLUDE directive specifies a device other than a disk.

User Action: Change the %INCLUDE directive to specify a random access device.

INCLUDE directive RMS error number <number>

Explanation: ERROR—A %INCLUDE directive caused an RMS–11 error when accessing the specified file.

User Action: Take action based on the reported RMS–11 error number.

INCLUDE directive syntax error

Explanation: ERROR—A %INCLUDE directive is not followed by a quoted string.

User Action: Supply a quoted string.

inconsistent function usage for function <name>

Explanation: ERROR—The parameter list in a DEF function invocation contains a string where the function expected a number, or vice versa. This message is issued only when the invocation occurs before the DEF statement in the program.

User Action: Supply a correct parameter in the function invocation or correct the parameter list in the DEF statement.

inconsistent subscript use for <array-name>

Explanation: ERROR—The number of subscripts in an array reference does not match the number of subscripts specified when the array was created.

User Action: Specify the same number of subscripts.

input prompt must be a string constant

Explanation: ERROR—An INPUT, LINPUT, or INPUT LINE argument list contains a numeric constant immediately following the statement.

User Action: Remove the numeric constant. You can specify only a string constant immediately after an INPUT, LINPUT, or INPUT LINE statement.

insufficient space for MAP DYNAMIC variable in MAP <name>

Explanation: ERROR—A variable named in a MAP DYNAMIC statement is larger than the space allocated in the corresponding MAP statement.

User Action: Increase the size of the MAP so it is large enough to hold the largest member.

integer constant exceeds machine integer size

Explanation: ERROR—The value specified in a DECLARE CONSTANT statement exceeds the largest allowable value for an integer. The maximum value is 32767.

User Action: Supply a value in the valid range.

integer constant required

Explanation: ERROR—The program contains a non-integer named constant in a context that requires an integer. For example:

DIM A ('123'D)

User Action: Supply an integer constant.

integer error or overflow

Explanation: WARNING—The program contains an integer expression whose value is outside the valid range.

User Action: Modify the expression so that its value is within the allowable range, or use an integer data type that can contain all possible values for the expression.

internal logic error detected

Explanation: ERROR—An internal logic error was detected.

User Action: This error should never occur. Submit a Software Performance Report with a machine-readable copy of the source program.

invalid conversion requested

Explanation: ERROR—The program contains a reference to the REAL or INTEGER function, and the argument to the function is an entire array or an RFA expression.

User Action: Remove the invalid argument. The argument to these functions must be a numeric expression.

invalid file format

Explanation: ERROR—The file is not a RSTS/E native mode file.

User Action: Put the file in the proper format. If it is an RMS–11 file, then use PIP to convert it to a RSTS/E native mode file.

invalid integer type

Explanation: ERROR—A reference to the INTEGER function contains an invalid data type keyword, for example, A = INTEGER(A, SINGLE).

User Action: Change the invalid data type keyword. The INTEGER function returns only BYTE, WORD, or LONG values.

invalid real type

Explanation: ERROR—A reference to the REAL function contains an invalid data type keyword, for example, A = REAL(A, LONG).

User Action: Change the invalid data type keyword. The REAL function returns only SINGLE, DOUBLE, GFLOAT, or HFLOAT values.

<command> is an illegal command from initialization file

Explanation: ERROR—An initialization file contains an invalid command, such as COMPILE.

User Action: Remove the invalid command.

<text> is an invalid keyword value

Explanation: FATAL—The command supplied an invalid value for a keyword.

User Action: Supply a valid value.

<clause> is an unsupported OPEN clause

Explanation: ERROR—An OPEN statement specifies invalid attributes for the file.

User Action: Substitute valid attributes for the file.

<name> is not a DYNAMIC MAP variable of MAP <name>

Explanation: ERROR—A REMAP statement names a variable that was not named in the MAP DYNAMIC statement for the associated MAP statement.

User Action: Remove the variable from the REMAP statement, or name the variable in the MAP DYNAMIC statement for the associated MAP statement.

ITERATE must appear within a loop

Explanation: ERROR—The program contains an ITERATE statement that is not within a FOR-NEXT, WHILE, or UNTIL loop.

User Action: Remove the ITERATE statement or surround it with a loop.

jump into DEF

Explanation: ERROR—The program attempts to transfer control into a DEF block.

User Action: Change the control statement; you cannot transfer control into a DEF block except by invoking the function.

jump out of DEF

Explanation: ERROR—The program attempts to transfer control out of a DEF block.

User Action: Change the control statement; you cannot transfer control out of a DEF block except by an EXIT DEF, FNEXIT, FNEND, or END DEF statement.

jump out of program unit

Explanation: ERROR—In a source file containing more than one program module, a statement attempts to transfer control from one module into another.

User Action: Change the statement that attempts to transfer control; you cannot transfer control into a different program module.

jump to label: <label> is into a block

Explanation: ERROR—The program attempted to transfer control into a FOR-NEXT, WHILE, UNTIL, or SELECT-CASE block.

User Action: Change the program logic so that it does not transfer control into a block.

jump to line number <number> is into a block

Explanation: INFORMATION—The program transfers control to a line number within a FOR-NEXT, WHILE, UNTIL, or SELECT-CASE block.

User Action: This is an informational message. However, it is bad programming practice to transfer control into a block.

jump to unreferencable line number <lin-num>

Explanation: ERROR—A RESUME, GOSUB, or GOTO statement attempts to transfer control to a CASE statement.

User Action: Label or number the SELECT statement and transfer control to the beginning of the SELECT-CASE block.

key <vbl-name> in MAP <map-name> cannot be a dynamic variable

Explanation: ERROR—A KEY clause in an OPEN statement specifies a variable declared as dynamic in a MAP DYNAMIC statement.

User Action: Specify a static variable in the KEY clause; that is, declare the variable in a MAP statement, not a MAP DYNAMIC statement.

KEY <vbl-name> in MAP <name> is too long (max is 255)

Explanation: ERROR—A KEY variable is longer than 255 characters.

User Action: Reduce the length of the KEY variable. The maximum key length is 255 characters.

KEY <vbl-name> is not an unsubscripted variable in MAP <name>

Explanation: ERROR—An OPEN statement for an indexed file specifies a KEY variable that does not appear in a MAP statement.

User Action: Place the KEY variable in the MAP statement referenced by the OPEN statement's MAP clause.

KEY clauses require a MAP clause

Explanation: ERROR—An OPEN statement specifies KEY clauses without specifying a MAP clause.

User Action: Supply a MAP clause to define the position of the keys in the record buffer.

key is needed for indexed files

Explanation: ERROR—The program attempts to open an indexed file for output, and the PRIMARY KEY clause is missing.

User Action: Supply a PRIMARY KEY clause.

key must be either integer or string

Explanation: ERROR—A FIND or GET statement on an indexed file contains a key specification that is not an integer or string.

User Action: Change the key specification to be an integer or a string.

key segment <vbl-name> in map <map-name> must be a string key

Explanation: ERROR—An OPEN statement specifies a segmented key containing a numeric variable. For example:

```
10   OPEN ''INDEX.DAT'' AS FILE #1, ORGANIZATION INDEXED,  &
          PRIMARY KEY (A$, B$, C%), MAP ABC
```

User Action: Specify only string variables in segmented keys.

key, <vbl-name> in map <map-name> must be either integer or string

Explanation: ERROR—An OPEN statement contains a key specification that is not an unsubscripted integer or string variable.

User Action: Change the key specification to be an unsubscripted integer or string variable.

keyword inconsistent with <OPEN clause> clause

Explanation: ERROR—An OPEN statement contains an ALLOW, ACCESS, or RECORDTYPE clause whose keyword argument is invalid, for example, ACCESS FORTRAN.

User Action: Change the clause argument to a valid keyword for that clause.

keyword <name> is reserved in VAX BASIC

Explanation: WARNING—This keyword is reserved in VAX BASIC.

User Action: Remove the keyword if you want your program to be transportable.

<keyword> keyword inconsistent with <keyword>

Explanation: ERROR—An OPEN statement contains contradictory record format specifications, for example, both FIXED and VARIABLE.

User Action: Specify only one record format.

<keyword> keyword is inconsistent with file organization

Explanation: ERROR—An OPEN statement contains a keyword that is inapplicable to the file organization, for example, ACCESS SCRATCH with ORGANIZATION VIRTUAL.

User Action: Remove the inconsistent keyword.

<text> keyword requires a value

Explanation: ERROR—A keyword command was typed without a value.

User Action: Supply a valid keyword value.

label <label> not defined

Explanation: ERROR—The program tries to transfer control to a non-existent label.

User Action: Define the label before transferring control to it.

label <name> does not label an active block statement

Explanation: ERROR—An EXIT statement in a loop, IF-THEN-ELSE, or SELECT-CASE block specifies a label which does not refer to that block.

User Action: Change the program so that the label actually refers to the block in which the EXIT statement occurs.

label <name> does not label an active loop statement

Explanation: ERROR—In a loop, an EXIT or ITERATE statement specifies a label which does not refer to that loop.

User Action: Change the program so that the label actually refers to the loop in which the EXIT or ITERATE statement occurs.

label not allowed on RESUME

Explanation: ERROR—A RESUME statement specifies a label rather than a line number.

User Action: Change the label to a line number.

language feature is declining

Explanation: INFORMATION—This error is reported only when the FLAG:DECLINING qualifier is in effect. The program contains a language feature that is not recommended for new program development, for example, the FIELD statement.

User Action: DIGITAL suggests that you use (1) MAP, MAP DYNAMIC, and REMAP statements instead of a FIELD statement, (2) EDIT$ rather than CVT$$ functions, and (3) overlaid MAPs rather than CVTxx functions.

language feature is operating system dependent

Explanation: ERROR—The program contains a PRINT statement with RECORD clause on a system which does not support the RECORD clause.

User Action: Remove the RECORD clause.

LET directive syntax error

Explanation: ERROR—A %LET directive contains a syntax error, for example, an invalid lexical identifier, or you specified the %LET directive while syntax checking was enabled.

User Action: Either use the correct syntax for the %LET directive or disable syntax checking.

lexical identifier must be declared before reference

Explanation: ERROR—A %IF directive names a lexical constant which was not named in a preceding %LET directive.

User Action: Declare the lexical constant with the %LET directive before referencing it.

line number <n> follows line number <m>

Explanation: WARNING—The source program line numbers are not in ascending order.

User Action: Put the line numbers in ascending order.

line number <n> undefined due to conditional compilation

Explanation: ERROR—The program references a line number which does not appear in the object code as a result of the branch taken in a %IF-%THEN-%ELSE-%END-%IF directive.

User Action: Change the %IF-%THEN-%ELSE-%END-%IF directive or remove the line number reference.

line number may not appear in INCLUDE directive file

Explanation: ERROR—The file specified in a %INCLUDE directive contains a line number.

User Action: Remove the line number from the file.

line too long

Explanation: ERROR—A program contains either (1) a text line that exceeds the maximum characters allowed (the maximum length of a text line is 225 characters on RSTS/E systems and 132 characters on RSX–11M /M–PLUS systems), or (2) a multi-statement line that contains more than 32767 characters.

User Action: Either (1) break the line into two text lines, using the ampersand (&) continuation character, or (2) split the multi-statement line by adding a new line number.

logical operation on non-integer quantity

Explanation: ERROR—The program contains a logical operation performed on strings or real numbers.

User Action: Change the logical operands to integers.

loop control variable must be a numeric variable

Explanation: ERROR—A FOR statement attempts to assign a string expression as the loop control variable's initial value.

User Action: Remove the string expression. You can assign only numeric values as the loop's initial value.

loop initial value must be a numeric expression

Explanation: ERROR—A FOR statement attempts to assign a string value to the loop control variable.

User Action: Remove the string expression. You can assign only numeric values to the loop control variable.

loop limit must be numeric

Explanation: ERROR—A FOR statement attempts to assign a string expression as the loop control variable's limiting value.

User Action: Remove the string expression. You can assign only numeric values as the loop control variable's limiting value.

loop will never execute

Explanation: WARNING—The program contains a FOR-NEXT loop that is not executable; for example, FOR I% = 1% TO 0%. Compilation continues, but the loop is ignored.

User Action: Change the loop parameters or insert an appropriate STEP clause.

MAP <name> larger than previously defined

Explanation: FATAL—When a program is run with the BASIC–PLUS–2 RUN command, the length of a map is defined by the length of the first occurrence of the map. Therefore, if the current program module in memory defines a MAP as 100 bytes in length and a loaded program module defines the same MAP as 50 bytes in length, the MAP is defined as containing 50 bytes (because the loaded module was compiled before the program module currently in memory).

User Action: Make sure the MAP statements are the same size.

MAP <name> used in OPEN not defined

Explanation: ERROR—An OPEN statement's MAP clause references a non-existent MAP statement.

User Action: Define the MAP statement referenced by the MAP clause, or remove the MAP clause.

MAP DYNAMIC <map-name> may not be larger than 32767 bytes

Explanation: ERROR—A MAP DYNAMIC statement references a map that is greater than 32767 bytes in size.

User Action: Reduce the size of the map, as defined in the MAP statement(s), to 32767 bytes or less.

MAP DYNAMIC <name> requires MAP or static string

Explanation: ERROR—The program contains a MAP DYNAMIC statement whose name does not appear in a MAP statement or is not a static string. Because parameters to DEF functions are only local copies, you cannot use a MAP DYNAMIC statement on a DEF parameter.

User Action: Provide a corresponding MAP statement with the same name or declare the string name within the preceeding MAP to make it a static string.

MAP statement requires map name

Explanation: ERROR—A MAP statement does not specify a map name.

User Action: Specify a name for the MAP.

MAP too large in OPEN

Explanation: ERROR—The size of the MAP area referenced in an OPEN statement is greater than 32767 bytes.

User Action: Reduce the size of the MAP area.

MAP variable <name> referenced before declaration

Explanation: INFORMATION—A reference to a MAP variable occurs before the MAP statement.

User Action: Make sure that the MAP statement precedes any references to variables in the MAP.

MAT statements require one or two dimensions

Explanation: ERROR—A MAT statement references an array of more than two dimensions.

User Action: Remove the array reference. MAT statements are valid only on arrays of one or two dimensions.

matrix dimension error

Explanation: ERROR—The program either (1) contains a MAT IDN, MAT TRN, or MAT INV performed on a one dimensional array, or (2) performs a matrix operation which requires identical subscripts in the operand arrays and those arrays have different subscripts.

User Action: Dimension the arrays to the proper number of subscripts.

maximum conditional compilation depth exceeded

Explanation: ERROR—There are too many nested %IF-%THEN-%ELSE-%END-%IF directives in the program.

User Action: Reduce the number of nested %IF-%THEN-%ELSE-%END-%IF directives.

maximum number of dimensions exceeded. Maximum is <number>

Explanation: ERROR—An array declaration specifies more than the allowed number of dimensions.

User Action: Reduce the number of dimensions. The maximum is 8.

maximum parameters exceeded for <name>. Maximum is <number>

Explanation: ERROR—The program attempts to declare a DEF statement with more than 32 parameters or a subprogram with more than 255 parameters.

User Action: Reduce the number of parameters; DEF statements allow up to 32 parameters and subprograms allow up to 255 parameters.

<item> may not be passed BY <mechanism>

Explanation: ERROR—The program specifies an incorrect passing mechanism for a parameter's data type or an invalid parameter. For example, you cannot pass an entire array BY VALUE, nor can you pass a label as a parameter.

User Action: Specify a valid parameter or passing mechanism.

mismatched END, expected <block>

Explanation: ERROR—The program contains an incorrect END statement, for example, an END RECORD statement instead of an END GROUP statement.

User Action: Supply the correct type of END statement.

missing END IF directive before end of program unit

Explanation: ERROR—A %IF directive crosses a program module boundary.

User Action: Terminate the %IF with a %END %IF before beginning a new source module.

mode for parameter <n> of routine <name> changed to match declaration

Explanation: ERROR—The data type specified in a routine invocation does not match that of the routine declaration. BASIC–PLUS–2 issues this message only if the data type conversion results in a parameter that cannot be modified by the routine that was invoked.

User Action: Make the data type specifications in the declaration and the invocation match.

mode for parameter <n> of routine <name> not as declared

Explanation: ERROR—The CALL or invocation of a routine specifies a string argument for a parameter that was specified as a numeric when the routine was declared, or vice versa.

User Action: Change the string parameter to numeric, or vice versa.

module <name> not a BASIC–PLUS–2 object module

Explanation: FATAL—A program module being loaded with the BASIC–PLUS–2 LOAD command was not created by BASIC–PLUS–2 Version 2.0.

User Action: If the module source file is written in an earlier verion of BP2, compile the source file with the BASIC–PLUS–2 COMPILE command and reload the module. If the source file was written in another language (MACRO, for example), link and run your program from the system monitor level.

more than one main module

Explanation: FATAL—There is more than one main program loaded, or a main module is loaded and another main module is the current program module when the RUN command is given.

User Action: Reload the program modules, removing all but one of the main modules.

multiple definition of <name>

Explanation: ERROR—A variable is declared in more than one declarative statement.

User Action: Make sure that the variable is declared only once.

multiple definition of lexical identifier is illegal

Explanation: ERROR—A lexical constant is named in more than one %LET directive.

User Action: Declare the lexical constant only once with %LET.

name is too long, changed to <name>

Explanation: WARNING—A variable or array name is longer than 31 characters. BASIC–PLUS–2 truncates the name to 31 characters and continues compilation so that other errors may be detected. The actual program line remains unchanged and no object file is produced.

User Action: Reduce the length of the variable name to 31 or fewer characters.

named array <array-name> is too large

Explanation: ERROR—An array requires more than $2^{29} - 1$ bytes of storage.

User Action: Reduce the size of the array.

negative FILL or string length

Explanation: ERROR—The program contains a negative FILL specification or string length.

User Action: Change the FILL specification or string length to a positive number.

nested FOR loops with same control variable <name>

Explanation: ERROR—The program contains nested FOR-NEXT loops that use the same index variable.

User Action: Change the index variable for all but one of the loops.

no change made

Explanation: WARNING—The search string in an EDIT command was not located in the text.

User Action: Enter valid search string.

no file specified for command in initialization file—command ignored

Explanation: WARNING—An initialization file contains a BRLRES, DSKLIB, LIBRARY, ODLRMS, or RMSRES command or qualifier without the required file specification.

User Action: Provide a valid file specification.

no main program

Explanation: FATAL—When the BASIC–PLUS–2 RUN command was given, at least one subprogram was loaded or currently in memory, but there was no main module loaded as an object module or currently in memory.

User Action: If there is no current program module in memory, call in the main module with the OLD command. Otherwise, compile and reload all subprograms as object modules and then call the main program into memory.

no such MAP area <name>

Explanation: ERROR—A REMAP statement names a non-existent MAP area.

User Action: Supply a MAP statement before executing the REMAP statement.

numeric array expected

Explanation: ERROR—A CHANGE statement does not specify a numeric array.

User Action: Supply a numeric array in the CHANGE statement.

numeric constant required

Explanation: ERROR—The program contains a string in a context that requires a numeric constant. For example:

```
DECLARE INTEGER CONSTANT A = "ABC"
```

User Action: Supply a numeric constant.

numeric expression is needed

Explanation: ERROR—The program contains a string expression in a context that requires a numeric expression, for example, WHILE A$.

User Action: Supply a numeric expression.

numeric expression is needed in built-in function

Explanation: ERROR—A reference to a BASIC–PLUS–2 built-in function contains a string instead of a numeric expression.

User Action: Supply a numeric expression.

object file I/O error # <num>

Explanation: FATAL—An error occured during the creation of the object file. The error number is the RMS returned status.

User Action: Take action based on the RMS error number.

OPEN clause <clause> value greater than <number>

Explanation: ERROR—An OPEN statement contains a RECORDSIZE, FILESIZE, EXTENDSIZE, WINDOWSIZE, BLOCKSIZE, BUCKETSIZE, or BUFFER clause whose argument is too large.

User Action: Supply a smaller value for the argument.

operator expected, not found

Explanation: A compiler directive contains an invalid lexical expression which has a right parenthesis immediately followed by a lexical identifier.

User Action: Correct the lexical expression.

operator must follow right parenthesis

Explanation: ERROR—The program contains an incorrect lexical expression.

User Action: Correct the lexical expression.

OPTION clause contradicts prior clause

Explanation: ERROR—The OPTION statement contains contradictory clauses, for example, specifying the default integer size as both BYTE and LONG.

User Action: Remove one of the clauses.

OPTION statement out of sequence

Explanation: ERROR—The OPTION statement is either (1) not the first statement in a main program or (2) not the first statement following the SUB or FUNCTION statement.

User Action: Move the OPTION statement so it is either the first statement in the main program or the first statement following the SUB or FUNCTION statement in the subprogram.

ORGANIZATION UNDEFINED requires FOR INPUT clause

Explanation: ERROR—The program opens a file with ORGANIZATION UNDEFINED, but does not specify FOR INPUT.

User Action: Specify FOR INPUT in the OPEN statement. You cannot write to a file with an undefined file organization. However, once you interpret the file organization using the FSP$ function, you can perform output by closing the file and then reopening it FOR OUTPUT, specifying the appropriate organization clause.

<keyword> overrides NOLINE

Explanation: WARNING—The program (1) was compiled with the NOLINES qualifier and (2) uses a keyword that requires line number information. For example, ERL and RESUME with line number statements both require that the program be compiled with the LINES qualifier.

User Action: None. If you use a keyword that requires line number information, BASIC–PLUS–2 automatically overrides the NOLINE default and sets LINES in effect.

parameter <n> of <type> structure not as declared

Explanation: ERROR—The actual parameter list in a CALL to a SUB subprogram or an invocation of a FUNCTION subprogram specifies an entire array where the subprogram declaration specified a simple variable, or vice versa.

User Action: Change the actual parameter list to match the declared parameter list, or vice versa.

parameter may not be received by <mechanism>

Explanation: ERROR—The subprogram specifies an incorrect passing mechanism for a parameter's data type or an invalid parameter. For example, you cannot receive an entire array BY VALUE.

User Action: Specify a valid parameter or passing mechanism.

parameter type specification required with /EXPLICIT

Explanation: ERROR—In a program compiled with the /TYPE:EXPLICIT qualifier, no data type keyword is specified for a parameter.

User Action: Supply a data type keyword for the parameter. There are no default data types when you compile a program with the /TYPE:EXPLICIT qualifier.

<n> parameters expected for <routine>

Explanation: ERROR—The CALL statement or invocation of a routine specifies a different number of parameters than the number specified when the routine was declared.

User Action: Change the number of parameters to match the number declared.

passing mechanism disagrees with declaration

Explanation: ERROR—The CALL statement or invocation of a routine specifies a different passing mechanism for a parameter than that specified when the routine was declared.

User Action: Remove the BY clause specified in the CALL statement or invocation; BASIC–PLUS–2 automatically passes parameters with the passing mechanism specified when the routine was declared.

passing mechanism not allowed for <item>

Explanation: ERROR—The program specifies a passing mechanism in a context other than an external subprogram invocation or declaration.

User Action: Remove the passing mechanism.

passing mechanism not allowed for DEF <vbl-name>

Explanation: ERROR—A DEF invocation specifies a passing mechanism for a parameter.

User Action: Remove the passing mechanism.

PRINT USING clause must be a string expression

Explanation: ERROR—A PRINT USING statement specifies a numeric format string.

User Action: Supply a valid format string.

PRINT USING conflicts with RECORD clause

Explanation: ERROR—A PRINT USING statement contains a RECORD clause.

User Action: Remove the RECORD clause or use the PRINT statement instead of PRINT USING.

program structures nested too deeply

Explanation: FATAL—The program contains too many nested block constructs, for example, DEF function definitions.

User Action: Reduce the number of nested block constructs.

program too big to compile

Explanation: FATAL—The program is too big.

User Action: Recode the program as two or more modules.

program too large to RUN

Explanation: FATAL—Maximum memory has been exceeded by a program being run in the BASIC–PLUS–2 environment.

User Action: To reduce the memory requirements of your program, write an overlay descriptor (ODL) file to overlay the program modules; then link and run your task from the system monitor level. (If you ran the program with the /DEBUG qualifier, the debugger added to the size of your task. You may be able to run the program in the BASIC–PLUS–2 environment with the RUN/NODEBUG command.)

radix not supported

Explanation: ERROR—A literal constant specifies a radix. For example, in the following DECLARE statement, H is an invalid radix specifier:

```
10    DECLARE LONG CONSTANT A = H"111"
```

User Action: Supply a valid BASIC–PLUS–2 radix specifier.

READ access inconsistent with FOR OUTPUT

Explanation: ERROR—An OPEN statement specifies the FOR OUTPUT and ACCESS READ clauses.

User Action: The FOR OUTPUT clause specifies that a new file is created; ACCESS READ specifies that the program can only read the file. If you want to create a new file, remove the ACCESS READ clause; if you want read-only access to a file, specify the FOR INPUT clause.

READ without DATA statement

Explanation: ERROR—The program contains a READ statement and there are no DATA statements.

User Action: Supply a DATA statement, or remove the READ statement.

real constant expressions not supported

Explanation: ERROR—The program contains a DECLARE REAL CONSTANT statement that specifies an expression for the constant value.

User Action: Remove the expression. You can specify only literal values when declaring floating-point constants.

record too big from INCLUDE directive file

Explanation: FATAL—The file specified in a %INCLUDE directive contains a record longer than 255 characters.

User Action: Edit the file to remove any records longer than 255 characters.

repeat count must be positive numeric

Explanation: ERROR—A FILL item specifies a non-numeric or negative repeat count, for example, FILL(A$) or FILL(–3).

User Action: Supply a valid repeat count.

<item> requires a numeric expression

Explanation: ERROR—The program contains a string expression in a context requiring a numeric expression.

User Action: Supply a numeric expression.

<item> requires conditional expression

Explanation: ERROR—A CASE or IF keyword is immediately followed by a floating-point or string expression.

User Action: Supply a conditional expression (relational, logical, or integer).

<item> requires string expression

Explanation: ERROR—The program contains a numeric expression in a context requiring a string expression, for example, the file specification in an OPEN statement, or the default file specification in a DEFAULTNAME clause.

User Action: Supply a string expression.

result attributes inconsistent with prior declaration

Explanation: ERROR—An external or DEF function declaration specifies a different data type for the function's result than the DEF or FUNCTION statement specifies.

User Action: Change the specified data type in either the declaration or the DEF or FUNCTION statement so that the data types agree.

RFA expression required

Explanation: ERROR—A GET statement's RFA clause contains an expression that is not of the RFA data type.

User Action: Supply a valid RFA expression.

RFA not allowed in this context

Explanation: ERROR—The program attempts to use an RFA expression in an arithmetic expression or other invalid context.

User Action: Remove the RFA expression. You can use the RFA data type only in file I/O, in an assignment statement or in a comparison.

scale factor has been set to <number>

Explanation: WARNING—A SCALE command has reset the scale factor.

User Action: None.

scale factor out of range—ignored

Explanation: WARNING—The SCALE qualifier specifies a scale factor that is not between 0 and 6, inclusive.

User Action: Supply a valid scale factor.

scale has been truncated to <number>

Explanation: WARNING—A floating-point number was specified in the SCALE command. The number has been truncated and the resulting integer is now the scale factor.

User Action: None.

scale is out of range—valid is 0 to 6

Explanation: ERROR—The OPTION statement specifies a scale factor that is not between 0 and 6, inclusive.

User Action: Supply a valid scale factor.

scale factor used is 0 for single precision

Explanation: WARNING—An attempt was made to set the SCALE factor while in single precision.

User Action: Set the precision to DOUBLE. You cannot use scaling when in single precision.

SINGLE constant required

Explanation: ERROR—The program contains a DECLARE SINGLE CONSTANT statement that specifies an expression for the constant value.

User Action: Remove the expression. You can specify only literal values when declaring floating-point constants.

SPAN is inconsistent with NOSPAN

Explanation: WARNING—An OPEN statement specifies both SPAN and NOSPAN clauses.

User Action: Remove one of the clauses.

specified numeric exceeds valid character code

Explanation: FATAL—A quoted literal of type character C contains a value outside the valid range, for example, '300'C.

User Action: Use a valid ASCII value.

star (*) is needed in DEF, not "/"

Explanation: ERROR—The program contains a statement that starts with DEF/.

User Action: Change the DEF/ to DEF*.

string constant expression is too long

Explanation: ERROR—The program contains a DECLARE STRING CONSTANT statement where the value assigned to the constant exceeds the maximum number of characters allowed for string constant expressions. In BASIC–PLUS–2, the maximum length of a string constant expression at compile time is 128 characters.

User Action: Change the string constant to a string variable and assign the string expression to the variable at run time.

string constant required

Explanation: ERROR—The program contains a numeric expression in a context that requires a string expression. For example:

```
DECLARE STRING CONSTANT ABC = 123
```

User Action: Supply a string literal or a named string constant.

string expression is needed

Explanation: ERROR—The program contains a numeric expression where a string expression is needed, for example, NAME 1% AS "ABC.DAT".

User Action: Supply a string expression.

string expression is needed in built-in function

Explanation: ERROR—The program specifies a numeric expression for a built-in function that requires a string argument.

User Action: Supply a string expression for the built-in function.

string is too large

Explanation: ERROR—A string exceeds the maximum allowable length. The maximum length is 32767 characters.

User Action: Reduce the length of the string.

string length not allowed on dynamic string <name>

Explanation: ERROR—The program contains a dynamic string variable declaration that specifies a string length.

User Action: Length specifications are allowed only for fixed-length strings; remove the length specification from the dynamic string, or allocate the string in a MAP or COMMON statement.

string length not allowed on MAP DYNAMIC variable

Explanation: ERROR—A string variable in a MAP DYNAMIC statement specifies a string length.

User Action: Remove the string length. All string variables named in a MAP DYNAMIC statement have a length of zero until a REMAP statement executes.

string length not allowed on numeric FILL

Explanation: ERROR—The program contains a numeric FILL item that specifies a length.

User Action: Remove the length specification from the numeric FILL item.

string length not allowed on numeric variable <name>

Explanation: ERROR—The declaration for a numeric variable contains a specification for string length.

User Action: Remove the string length specification.

string length specification for <name> must be numeric

Explanation: ERROR—The length specification for a fixed-length string is non-numeric, for example COMMON A$ = "ABC".

User Action: Supply a numeric length specification.

string literal required for compiler directive

Explanation: ERROR—A quoted string is missing in a compiler directive that requires one, for example, %IDENT.

User Action: Supply a string literal for the compiler directive.

string variable expected

Explanation: ERROR—A CHANGE statement specifies a numeric variable.

User Action: Supply a string variable; the CHANGE statement changes a string variable to a numeric array, and vice versa.

string variable required

Explanation: ERROR—A statement references a numeric variable instead of a string variable, for example LINPUT A%.

User Action: Supply a string variable instead of a numeric variable.

subscript may not be specified for entire array

Explanation: ERROR—A CALL statement or external function reference passes an entire array as a parameter and contains a subscript expression, for example A(,,3).

User Action: Remove the subscript expression. You cannot specify any subscripts when passing an entire array as a parameter.

subscript out of range for <array-name>

Explanation: ERROR—The program references an array element with constant subscript(s) outside the bounds of the array.

User Action: Check program logic to make sure all subscripts are within the bounds of the array.

suffix not allowed on FILL after data-type keyword

Explanation: ERROR—A FILL item defined with an explicit data type ends in a percent or dollar sign.

User Action: Remove the FILL item's percent or dollar sign.

suffix not allowed on variable <name>

Explanation: ERROR—A variable defined with explicit data type ends in a percent or dollar sign.

User Action: Remove the variable's percent or dollar sign.

symbol <name> multiply defined

Explanation: FATAL—More than one subprogram with the same name has been loaded with the BASIC–PLUS–2 LOAD command.

User Action: Remove or rename the subprograms.

system commands cannot be executed from ini file

Explanation: ERROR—An initialization file contains a command preceded by a dollar sign.

User Action: Remove the command from the initialization file. Initialization files cannot perform system commands.

text following END ignored

Explanation: ERROR—The compiler detected text following an END, END SUB, or END FUNCTION statement.

User Action: Remove the text. If an END, END SUB, or END FUNCTION statement appears in the program, it must be the last statement.

THEN directive must follow a lexical expression

Explanation: ERROR—A %IF directive contains a lexical expression that is not immediately followed by a %THEN.

User Action: Supply a %THEN clause. %THEN, %ELSE, and %END %IF are required in a %IF directive.

too few arguments

Explanation: ERROR—The invocation of a BASIC–PLUS–2 built-in function contains too few arguments.

User Action: Supply the correct number of arguments to the function.

too many arguments

Explanation: ERROR—The invocation of a BASIC–PLUS–2 built-in function contains too many arguments.

User Action: Supply the correct number of arguments to the function.

too many array indices active

Explanation: ERROR—A subscript expression contains more than 100 array indices between the open parenthesis and the close parenthesis.

User Action: Reduce the number of active array indices.

too many function parameters active

Explanation: ERROR—An external function invocation contains too many expressions in the actual parameter list.

User Action: Reduce the number of expressions in the actual parameter by assigning the expressions to temporary variables.

too many keys—limit is 255

Explanation: ERROR—An OPEN statement specifies more than 255 index keys.

User Action: Reduce the number of index keys. The maximum is 255.

too many subprograms or maps

Explanation: FATAL—A program being run in the BASIC–PLUS–2 environment contains more than 16 MAPs or calls more than 16 subprograms.

User Action: Link and run your program from the system monitor level.

too many temporaries generated for DEF at line <line-number> statement <statement-number>

Explanation: ERROR—A program contains code within a DEF function that contains too large a number of temporary variables.

User Action: Either change the DEF function to an external function or reduce the number of temporaries required within the DEF function.

TYPE default of STRING is not allowed

Explanation: ERROR—STRING was specified as the default data type in (1) a compiler command, (2) a qualifier to the BASIC–PLUS–2 DCL command, or (3) an OPTION statement.

User Action: Specify a numeric data type as the default.

unable to copy file, RMS error <error number>

Explanation: FATAL—An RMS–11 error occurred while attempting to copy the RUN task into the user's account. See the *RSTS/E RMS–11 User's Guide* or the *RSX–11M/M–PLUS RMS–11 User's Guide* for an explanation of the error.

User Action: Take action based on the associated RMS error.

unaligned COMMON or MAP variable <vbl-name> in <psect>

Explanation: WARNING—The total storage preceding a numeric variable in a COMMON or MAP is an odd number of bytes.

User Action: None. BASIC–PLUS–2 pads the preceding storage with a blank byte because the PDP–11 requires that numeric data start on a word boundary. VAX BASIC does not pad the storage because numeric data can start on any byte boundary. However, if you want the program to run on both types of system, you should ensure that all numeric data start on a word boundary.

undefined line number

Explanation: ERROR—A statement tries to transfer control to a non-existent line.

User Action: Replace the non-existent line number with the correct destination line number.

undefined/unresolved global <name>

Explanation: FATAL—Either (1) a call was made to a subprogram that was not a loaded module or the current program module in memory when the RUN command was given, or (2) the compiler generated a global that does not exist in the RUN task.

User Action: (1) Make sure all external subprograms are either loaded or are the current program in memory. (2) If all subprograms are loaded or currently in memory and this message appears, submit an SPR and include all relevant information.

unexpected end of file

Explanation: ERROR—The compiler encountered an end-of-file immediately after an ampersand continuation character.

User Action: Remove the ampersand continuation character, or continue the line.

unresolved/undefined symbols

Explanation: ERROR—A program executed in the BASIC–PLUS–2 environment calls or invokes a subprogram or routine that has not been loaded with the LOAD command.

User Action: Load the subprogram or routine before running the program in the BASIC–PLUS–2 environment.

unsaved change has been made, CTRL/Z or EXIT to exit

Explanation: WARNING—A BASIC–PLUS–2 source program in memory has been modified, and an EXIT command or Ctrl/Z has been entered. BASIC–PLUS–2 signals the error notifying you that if you exit from the compiler, the program modifications will be lost.

User Action: If you want to save the program, enter the SAVE command. If you do not want to save the program, enter EXIT or press Ctrl/Z.

unterminated string literal

Explanation: ERROR—The program contains an improperly terminated string literal; for example, "ABC, "ABC', and 'ABC" are all improperly terminated.

User Action: Use the same type of quotation mark (either single or double) for both beginning and ending string delimiters.

user ABORT directive <text>

Explanation: FATAL—The compilation was terminated as the result of a %ABORT directive. The compiler prints the text following the %ABORT directive.

User Action: None.

user PRINT Message: <text>

Explanation: INFORMATION—This message was generated as a result of a %PRINT directive. The compiler prints the text you specify in the context of this message.

User Action: None.

user variable <name> not allowed in declaration

Explanation: ERROR—The parameter list in an external subprogram declaration contains a user variable name.

User Action: Remove the variable from the parameter list. When declaring a routine, the parameter list can contain only data type and parameter-passing mechanism specifications.

value too large for constant

Explanation: ERROR—The value of an EXTERNAL CONSTANT is larger than the specified data type allows.

User Action: Make sure the data type specified in the EXTERNAL CONSTANT statement matches that of the actual constant.

variable <name> not aligned in COMMON/MAP <name>

Explanation: WARNING—The total storage preceding a numeric variable in a COMMON or MAP is an odd number of bytes.

User Action: None. BASIC–PLUS–2 pads the preceding storage with a blank byte because the PDP–11 requires that numeric data start on a word boundary. VAX BASIC does not pad the storage because numeric data can start on any byte boundary. However, if you want the program to run on both types of system, you should ensure that all numeric data start on a word boundary.

variable <name> not aligned in multiple references in MAP <name>

Explanation: ERROR—More than one overlaid MAP area contains the same variable, but the variable's position differs in the MAP statements.

User Action: The same variable can appear in multiple overlaid MAPs, but the variable must occupy the same position in the PSECT; make sure that the variable appears in the same position in the MAP statements.

variable or constant required

Explanation: ERROR—The program contains an executable DIM statement that contains an expression in the bounds list.

User Action: Remove the expression from the bounds list. Executable DIM statements can have only constants or variables (simple or subscripted) as bounds.

virtual array space exceeded at array <name>

Explanation: ERROR—The storage for virtual arrays on a single channel exceeds 2147483647 bytes.

User Action: If there is only one virtual array on the channel, you must reduce the amount of storage used by the array. However, if there is more than one virtual array on the channel, you can put each array on a separate channel.

virtual array string <name> length increased from <n> to <m>

Explanation: WARNING—In a string virtual array DIM statement, the specified string length is not a power of two.

User Action: None. BASIC–PLUS–2 increases the string length to the next higher power of two.

virtual array string <name> length truncated from <n> to <m>

Explanation: WARNING—A string virtual array specifies a string length greater than 512. BASIC–PLUS–2 truncates the length specification to 512.

User Action: None. The maximum string length for virtual arrays is 512.

WINDOWSIZE inconsistent with CLUSTERSIZE

Explanation: ERROR—An OPEN statement contains both a WINDOWSIZE and CLUSTERSIZE clause.

User Action: Remove either the WINDOWSIZE or the CLUSTERSIZE clause. CLUSTERSIZE is valid only on RSTS/E systems, and WINDOWSIZE is valid on all systems except RSTS/E.

work file error—out of space, program too large

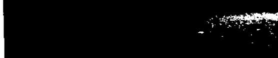
Explanation: FATAL—The BASIC–PLUS–2 workfile is full and the compilation can not continue because the program is too large to compile.

User Action: Recode the program as two or more modules.

work file error—RMS create failure

Explanation: FATAL—The BASIC–PLUS–2 workfile cannot be created.

User Action: Verify that the default account is accessible and that enough disk space is available to create the workfile.



B
Run-Time Error Messages

This appendix describes run-time errors, their causes, and the user action required to correct them.

B.1 Diagnosing Run-Time Errors

BASIC–PLUS–2 signals a run-time error message if an error occurs during program execution. There are three different levels of run-time errors: warning, error (also called trappable errors), and fatal. These errors are described as follows:

- Warning level errors

  Errors that have a warning severity level do not cause a program to abort; however, they do indicate that an error has occurred. In some cases, when a warning error occurs, BASIC–PLUS–2 prompts you for information or correct data; in other cases, program execution continues but the results are not as expected. You do not need error-handling routines to trap warning errors.

- Error level errors

  Errors that have an error severity level cause the program to abort unless they are trapped in a user-written error handling routine. Once the error is successfully handled, program execution continues.

- Fatal level errors

  Errors that have a fatal severity level cause the program to abort. You cannot trap fatal errors.

B.2 Error Message Format

The format of a BASIC–PLUS–2 run-time error message is as follows:

< l ><message> at line *n* in "module *module-name*"

< l > Is a character indicating the severity of the error. The severity indicator can be either a percent sign (%) or question mark (?). A question mark indicates an error of error or fatal severity. A percent sign indicates an error of warning severity.

<x> Is the line number where the error occurred.

<y> Is the name of the module where the error occurred.

B.3 Numerical List of Error Messages

This section contains a list of the BASIC–PLUS–2 run-time error messages. They are listed according to error number; from lowest to highest. (The first two errors listed are unnumbered.)

?Cannot open error file

Explanation: The BASIC–PLUS–2 error message file could not be opened.

User Action: On RSX systems, re-install BASIC–PLUS–2 to make sure that the error message file is successfully installed on the system. If this error still appears after a successful installation, please submit an SPR.

?RMS error # <num>

Explanation: RMS returned an error status that could not be handled by BASIC–PLUS–2. The number is the RMS error status returned by RMS.

User Action: See the RMS documentation for your system for a description of the error. If the error is –912, it is likely that you closed a file that other files were accessing with the CONNECT clause. If you close a file, you must also close all files connected to the file.

1 ?Bad directory for device

Explanation: ERROR—Either the device directory does not exist or is unreadable, or on RSTS/E systems, the program tried to access a magnetic tape with a file structure other than the default file structure.

User Action: Either supply a valid directory, change the default magnetic tape file structure with an ASSIGN MT0:.DOS or .ANSI command, or override the default by specifying MODE 16384 for DOS format or MODE 24576 for ANSI format when you open the file.

2 ?Illegal file name

Explanation: ERROR—The file name (1) is too long, (2) is incorrectly formatted, (3) contains embedded blanks or invalid characters, or (4) on RSX–11M/M–PLUS systems, is in lowercase letters.

User Action: Supply a valid file specification.

3 ?Account or device in use

Explanation: ERROR—The specified operation cannot be performed because the device is already in use or because the account cannot be found or opened.

User Action: Wait until the device is available or specify another device or account.

4 ?No room for user on device

Explanation: ERROR—No user storage space exists on the specified device or on RSTS/E systems. You have exceeded the number of files allocated for your account.

User Action: Delete files that are no longer needed to free disk space in the account in which the error occurred.

5 ?Can't find file or account

Explanation: ERROR—The specified file or directory is not on that device.

User Action: Supply a valid file specification.

6 ?Not a valid device

Explanation: ERROR—The device is illegal or nonexistent.

User Action: Supply a valid device.

7 ?I/O channel already open

Explanation: ERROR—The program attempts to open an I/O channel that is already open for input or output.

User Action: Close the channel and reopen it or specify another channel.

8 ?Device not available

Explanation: ERROR—The requested device is in use.

User Action: Wait until the device is available or specify a different device.

9 ?I/O channel not open

Explanation: ERROR—The program attempted to perform an I/O operation before opening the channel.

User Action: Open the channel with the OPEN statement before attempting an I/O operation to it.

10 ?Protection violation

Explanation: ERROR—The program attempted to read or write to a file whose protection code did not allow the operation, or on RSTS/E systems, you attempted to extend a contiguous file beyond its initially allocated size.

User Action: Use a different file, change the file's protection code, or change the attempted operation. On RSTS/E systems, use the FILESIZE clause with the CONTIGUOUS clause to allocate enough contiguous disk space when you open the file.

11 ?End of file on device

Explanation: ERROR—The program attempted to read data beyond the end of the file.

User Action: None. The program can trap this error in an error handler.

12 ?Fatal system I/O failure

Explanation: ERROR—An I/O error has occurred in either (1) the system or (2) Record Management Services. As a result, the last operation will not be completed.

User Action: See the *RSTS/E RMS–11 User's Guide* or the *RSX–11M/M–PLUS RMS–11 User's Guide* for information on RMS–11 errors, or retry the operation.

13 ?User data error on device

Explanation: ERROR—One or more characters may have transmitted incorrectly because of a parity error, bad punch combination on a card, or similar error.

User Action: Repeat the data entry operation.

14 ?Device hung or write locked

Explanation: ERROR—The program attempted an operation to a hardware device that is not functioning properly or is protected against writing.

User Action: Check the device on which the operation is performed.

15 ?Keyboard wait exhausted

Explanation: ERROR—No input was received during the execution of an INPUT, LINPUT, or INPUT LINE statement that was preceded by a WAIT statement.

User Action: None. You must supply input within the specified time.

16 ?Name or account now exists

Explanation: ERROR—The program attempted to create or rename a file or account with a file name that already exists.

User Action: Either (1) use the KILL statement to erase the old file before creating the file, or (2) use a different file name.

17 ?Too many open files on unit

Explanation: ERROR—The program attempted more than one DECtape output file per DECtape drive. BASIC permits only one open file per DECtape drive.

User Action: Close the file that is open on the unit or specify a different unit.

18 ?Illegal SYS( ) usage

Explanation: ERROR—RSTS/E only. The program attempted an illegal SYS call.

User Action: See the appropriate RSTS/E SYS call documentation.

19 ?Disk block is interlocked

Explanation: ERROR—RSTS/E only. The requested disk block segment is already in use (locked).

User Action: Try the operation again.

20 ?Pack IDs don't match

Explanation: ERROR—RSTS/E only. You have specified an incorrect identification code for the disk pack.

User Action: Use the correct pack ID.

21 ?Disk pack is not mounted

Explanation: ERROR—RSTS/E only. No disk pack is mounted on the specified disk drive.

User Action: Mount a disk pack on the disk drive or specify a disk drive that has a mounted disk pack.

22 ?Disk pack is locked out

Explanation: ERROR—RSTS/E only. The specified disk pack is mounted but is temporarily disabled.

User Action: Wait until the disk pack is available or specify another disk drive.

23 ?Illegal cluster size

Explanation: ERROR—RSTS/E only. The specified cluster size is unacceptable.

User Action: Change the cluster size. The cluster size must be a power of 2. A file cluster size must be equal to or greater than the pack cluster size and cannot be greater than 256. A pack cluster size must be equal to or greater than the device cluster size and cannot be greater than 16. The device cluster size is determined by the device type.

24 ?Disk pack is private

Explanation: ERROR—RSTS/E only. The program cannot access the specified disk pack.

User Action: Specify another disk pack.

25 ?Disk pack needs REBUILDing

Explanation: WARNING—RSTS/E only. The storage allocation table needs to be rebuilt and a nonfatal disk mounting error has occurred.

User Action: Use the CLEAN or ONLCLN operation in the UTILTY program.

26 ?Fatal disk pack mount error

Explanation: ERROR—RSTS/E only. The disk cannot be successfully mounted.

User Action: See your system manager.

27 ?I/O to detached keyboard

Explanation: ERROR—RSTS/E only. A program attempted to perform I/O to a hung-up dataset or to a detached console keyboard.

User Action: None. You cannot perform I/O to a detached keyboard or hung-up dataset.

28 ?Programmable ^C trap

Explanation: ERROR—A CTRL/C was pressed at the controlling terminal.

User Action: None. However, you can trap this error with an error handler.

29 ?Corrupted file structure

Explanation: ERROR—Either (1) RMS–11 has detected an invalid file structure on disk, or (2) on RSTS/E systems, a fatal error in a CLEAN operation has occurred.

User Action: See your system manager or the *RSTS/E RMS–11 User's Guide* or the *RSX–11M/M–PLUS RMS–11 User's Guide*.

30 ?Device not file-structured

Explanation: ERROR—A program attempted to access a nondisk device that is not file-structured. This error occurs, for example, when you try to gain a directory listing for a nondirectory device.

User Action: None. You cannot access a nondisk device that is not file-structured.

31 ?Illegal byte count for I/O

Explanation: ERROR—Either the program contains a PUT statement with a COUNT value greater than the RECORDSIZE clause established in the OPEN statement or the buffer specified in the MAP statement, or on RSTS/E systems, the disk is corrupted for your program if this error occurs when you try to execute the program.

User Action: Reduce the size of the COUNT clause to match the size of the buffer. You cannot put more characters in a buffer than the buffer's default or specified size.

32 ?No buffer space available

Explanation: ERROR—RSTS/E only. No buffer is available for file access. Possible causes are either (1) the receiving program has exceeded the pending message limit, or (2) the sending program has attempted to send a message and no small buffer is available for the operation.

User Action: See your system manager.

33 ?Odd address trap

Explanation: FATAL—Either (1) the program attempted to address nonexistent memory, or (2) on RSTS/E systems, the program attempted to access an odd address using the PEEK function.

User Action: None. Submit an SPR if this message appears for any reason other than those listed in the explanation and include all relevant output.

34 ?Reserved instruction trap

Explanation: FATAL—The program tried to execute an illegal or reserved instruction or a Floating-Point Processor (FPP) instruction on a system that does not have floating-point hardware.

User Action: None. If your system has floating-point hardware and this message appears, submit an SPR including all relevant information.

35 ?Memory management violation

Explanation: FATAL—Either (1) the program attempted to read or write to a memory location that does not allow access, or (2) on RSTS/E systems, the program attempted to access an address using the PEEK function.

User Action: None. Submit an SPR if this message appears for any reason other than those listed in the explanation and include all relevant output.

36 ?SP Stack Overflow

Explanation: ERROR—The program attempted to extend the program stack beyond its legal size.

User Action: None. If your program generates this message, submit an SPR and include all relevant information.

37 ?Disk error during swap

Explanation: ERROR—RSTS/E only. The system swapped your job into or out of memory. The contents of your job (the current task) are lost, but the job remains logged in to the system and control is returned to the keyboard monitor.

User Action: Report the occurrence of this error message to your system manager.

38 ?Memory parity failure

Explanation: ERROR—RSTS/E only. The memory occupied by your program has a parity error.

User Action: Contact your system manager.

40 ?Magtape record length error

Explanation: ERROR—RSTS/E only. A record in a file on magnetic tape was longer than the buffer designed to handle it.

User Action: Reduce the size of your record or increase the size of the buffer.

42 ?Virtual buffer too large

Explanation: ERROR—The program attempted to access a VIRTUAL file, and the buffer size was not a multiple of 512 bytes.

User Action: Change the I/O buffer to be a multiple of 512 bytes.

43 ?Virtual array not on disk

Explanation: ERROR—The program attempted to reference a virtual array on a nondisk device.

User Action: Virtual arrays must be on disk; change the file specification in the OPEN statement to open this array on disk.

44 ?Matrix or array too big

Explanation: ERROR—The program contains an array that is too large for memory.

User Action: Dimension the array with smaller subscripts.

45 ?Virtual array not yet open

Explanation: ERROR—The program attempted to reference a virtual array before the associated file on disk was opened.

User Action: Open the file on disk that contains the virtual array before you reference it.

46 ?Illegal I/O Channel

Explanation: ERROR—The program specified an I/O channel outside the legal range.

User Action: Specify I/O channels in the range 0 to 12, inclusive.

47 ?Line too long

Explanation: ERROR—The input line was longer than the record buffer.

User Action: Reduce the size of the input line to 255 characters.

48 %Floating point error

Explanation: WARNING—A program operation resulted in a floating-point number with an absolute value outside the range 10^{--38} to 10^{38}. If the program does not transfer to an error handling routine, BASIC returns a zero as the floating-point value for a number lower than 10^{--38} and the system's maximum positive number for a number higher than 10^{38}.

User Action: Check program logic or trap the error in an error handler.

50 %Data format error

Explanation: ERROR—The value supplied for a numeric variable is not a valid number, for example "ABC" and "1..2".

User Action: Supply numeric values of the correct data type.

51 %Integer error

Explanation: WARNING—The program requires an integer conversion from a larger data type (LONG or WORD) to a smaller data type (BYTE) and the resultant value is outside the allowable range. If the program does not transfer to an error handling routine, BASIC returns zero for the integer value.

User Action: Use an integer in the valid range. BYTE integers cannot be greater than 127; WORD integers cannot be greater than 32767; LONG integers cannot be greater than 2147483647.

52 ?Illegal number

Explanation: WARNING—The program specifies a data type in an INPUT or READ statement that does not agree with the value supplied. The number entered is too large for the desired variable.

User Action: Change the INPUT or READ statement or supply data of the correct type.

53 %Illegal argument in log

Explanation: WARNING—The program contains a negative or zero argument to the LOG or LOG10 function.

User Action: Supply an argument in the valid range.

54 %Imaginary square roots

Explanation: WARNING—An argument to the SQR function is negative. If the program does not transfer to an error handling routine, BASIC returns the square root of the absolute value of the argument.

User Action: Supply arguments to the SQR function that are greater than or equal to zero.

55 ?Subscript out of range

Explanation: ERROR—The program attempts to reference an array element outside of the array's dimensioned bounds.

User Action: Check program logic to make sure that all array references are to elements within the array boundaries.

56 ?Can't invert matrix

Explanation: ERROR—The program attempts to invert a single-dimensional array.

User Action: Supply a matrix array in the proper form for inversion.

57 ?Out of data

Explanation: ERROR—A READ statement requested additional data from an exhausted DATA list.

User Action: Remove the READ statement, reduce the number of variables in the READ statement, or supply more DATA items.

58 ?ON statement out of range

Explanation: ERROR—The index value in an ON GOTO or ON GOSUB statement is less than one or greater than the number of line numbers in the list.

User Action: Check program logic to make sure that the index value is greater than or equal to one, and less than or equal to the number of line numbers in the ON GOTO or ON GOSUB statement.

59 ?Not enough data in record

Explanation: ERROR—You did not supply enough data to fill all the specified variables in an INPUT statement.

User Action: Supply enough data or reduce the number of specified variables.

60 ?Integer overflow, FOR loop

Explanation: ERROR—The value of the loop index in the program exceeded the range for the loop variable. This can also occur during the evaluation of loop termination in the following cases:

- If the initial value minus the step value causes an overflow

- If the limit value plus the step value causes an overflow

User Action: Correct the loop index so it does not exceed the allowed range for the index's data type. You can do this by either modifying the loop or by using a different data type for the index variable.

61 %Division by 0

Explanation: WARNING—The program attempts to divide a value by zero. If the program does not transfer to an error handling routine, BASIC returns the value of zero.

User Action: Check program logic and change the attempted division, or trap the error in an error handler.

63 ?Field overflows buffer

Explanation: ERROR—A FIELD statement attempts to access more data than exists in the specified buffer.

User Action: Change the FIELD statement to match the buffer's size or increase the buffer's size.

64 ?Not a random access device

Explanation: ERROR—The program attempts a random access on a device that does not allow such access. This error occurs, for example, if you attempt a PUT operation with a RECORD clause to a file on magnetic tape.

User Action: Change the access to sequential instead of random or use a suitable I/O device.

65 ?Illegal MAGTAPE( ) usage

Explanation: ERROR—The program contains an incorrectly formatted or invalid MAGTAPE function code or function argument.

User Action: Change the MAGTAPE function code or argument.

72 ?RETURN without GOSUB

Explanation: FATAL—The program executes a RETURN statement before a GOSUB statement.

User Action: Check program logic to make sure that the RETURN statement is executed only in a subroutine or remove the RETURN statement.

73 ?FNEND without function call

Explanation: FATAL—The program attempts to execute an END DEF or FNEND statement before executing a function call.

User Action: Check program logic to make sure that the FNEND statement is executed only in a multi-line DEF or remove the END DEF or FNEND statement.

88 ?Arguments don't match

Explanation: FATAL—The arguments in a function call do not match the arguments defined for the function, either in number or in type.

User Action: Change the arguments in the function call to match those in the DEF statement or change the arguments in the DEF statement.

89 ?Too many arguments

Explanation: FATAL—A function invocation or CALL statement passed more arguments than were expected.

User Action: Reduce the number of arguments to the number expected. The maximum number of arguments is eight.

97 ?Too few arguments

Explanation: FATAL—A function invocation or CALL passed fewer arguments than were defined in the function or subprogram.

User Action: Change the number of arguments to match the number defined in the function or subprogram.

103 ?Program lost-Sorry

Explanation: FATAL—A fatal system error caused your program to be lost.

User Action: This error should never occur. Submit a Software Performance Report if this error occurs.

104 ?RESUME and no error

Explanation: FATAL—The program executes a RESUME statement outside of the error handling routine.

User Action: Check program logic to make sure that the RESUME statement is executed only in the error handler.

105 ?Redimensioned array

Explanation: FATAL—A matrix statement has tried to redimension an array larger than the array's initial allocation.

User Action: Do not redimension an array larger than its initial allocation. Correct the matrix statement.

116 ?PRINT-USING format error

Explanation: ERROR—The program contains a PRINT USING statement with an invalid format string.

User Action: Change the PRINT USING format string.

126 ?Maximum memory exceeded

Explanation: FATAL—The program has insufficient string and I/O buffer space because either (1) its allowable memory size has been exceeded, or (2) the system's maximum memory capacity has been reached.

User Action: Reduce the amount of string or I/O buffer space, or split the program into two or more modules.

127 %SCALE factor interlock

Explanation: FATAL—A subprogram was compiled with a different SCALE factor than the calling program.

User Action: Recompile one of the programs with a scale factor that matches the other.

128 ?Tape records not ANSI

Explanation: ERROR—The records on the magnetic tape you accessed are in neither ANSI D nor ANSI F format.

User Action: On RSX–11M/M–PLUS systems, remount the tape with the DOS (DO), Files–11 (RS) or RT–11 (RT) qualifier to determine the format of the records on the magnetic tape. On RSTS/E systems, set the magnetic tape to DOS format with the ASSIGN command or OPEN the file with the MODE 16384 (DOS) clause.

129 ?Tape BOT detected

Explanation: ERROR—The program attempts a rewind or backspace operation on a magnetic tape that is already at the beginning of the tape.

User Action: Check program logic; do not rewind or backspace if the magnetic tape is at its beginning.

130 ?Key not changeable

Explanation: ERROR— An UPDATE statement attempted to change a KEY field that did not have the CHANGES clause specified in the OPEN statement.

User Action: Specify the CHANGES clause for that key field in the OPEN statement. Note that the primary key cannot be changed and that you cannot specify the CHANGES clause when you open an existing file if the OPEN statement that created the file did not contain the CHANGES clause.

131 ?No current record

Explanation: ERROR— The program attempts a DELETE or UPDATE operation when the previous GET or FIND operation failed or when no previous GET or FIND operation was done.

User Action: Correct the cause of failure for the previous GET or FIND operation or make sure a GET or FIND operation was done, and then retry the operation.

132 ?Record has been deleted

Explanation: ERROR—A record previously located by its Record File Address (RFA) has been deleted.

User Action: None.

133 ?Illegal usage for device

Explanation: ERROR—The requested operation cannot be performed for the following reasons:

- The device specification contains illegal syntax.
- The specified device does not exist on your system.
- The specified device is inappropriate for the requested operation (for example, trying to access an INDEXED file on a magnetic tape).

User Action: Supply the correct device type.

134 ?Duplicate key detected

Explanation: ERROR— In a PUT operation to an indexed file, a duplicate key was specified, and the DUPLICATES clause was not specified when the file was created.

User Action: Change the duplicate key or recreate the file specifying the DUPLICATES clause for that key.

135 ?Illegal usage

Explanation: ERROR—The program tried to open either a file of undeclared organization or a file without a record operation specified in the ACCESS clause.

User Action: Declare the file organization in the OPEN statement or specify the record operation you want to perform in the ACCESS clause.

136 ?Illegal or illogical access

Explanation: ERROR—The requested access is impossible for the following reasons:

- The attempted record operation and the ACCESS clause in the OPEN statement are incompatible.
- The ACCESS clause is inconsistent with the file organization.
- The ACCESS READ or APPEND clause was specified when the file was created.

User Action: Change the ACCESS clause.

137 ?Illegal key attributes

Explanation: ERROR—The program specified an illegal combination of key characteristics.

User Action: Check the OPEN statement for either a NODUPLICATES clause and CHANGES clause, or a CHANGES clause without a DUPLICATES clause.

138 ?File is locked

Explanation: ERROR—The program does not allow shared access and attempts to access a file that has been locked by another user or by the system.

User Action: Change the ACCESS or ALLOW clause in the OPEN statement to allow shared access or wait until the file is released by other user(s).

139 ?Invalid file options

Explanation: ERROR—The program has specified invalid file options in the OPEN statement.

User Action: Change the invalid file options.

140 ?Index not initialized

Explanation: ERROR—The program attempts a GET or FIND operation on a record in an empty INDEXED file.

User Action: None. You cannot perform GET or FIND operations on a record that does not exist.

141 ?Illegal operation

Explanation: ERROR— The program attempts to do one of the following:

- Delete a record in a sequential file
- Update a record on a magnetic tape file
- Perform RMS–11 I/O on a virtual file (RSTS/E only)

User Action: Change the illegal operation. Block I/O requires virtual organization. RMS–11 I/O requires sequential, relative, or indexed organization.

142 ?Illegal record on file

Explanation: ERROR—A record contains an invalid record length.

User Action: Check the file for possible bad data.

143 ?Bad record identifier

Explanation: ERROR—The program attempted a record access that specified one of the following:

- A zero or negative record number on a RELATIVE file
- A GET or FIND operation on an INDEXED file with a null key

User Action: Change the record number or key specification to a valid value.

144 ?Invalid key of reference

Explanation: ERROR—The program attempted to perform a GET, FIND, or RESTORE operation on an INDEXED file using an invalid KEY clause, for example, an alternate KEY that has not been defined.

User Action: Use a valid KEY clause in the GET, FIND, or RESTORE statement.

145 ?Key size too large

Explanation: ERROR—The key length on a GET or FIND is either zero or larger than the key length defined for the target record.

User Action: Change the key specification in the GET or FIND statement.

146 ?Tape not ANSI labeled

Explanation: ERROR—The program attempts to access a file-structured magnetic tape that does not have an ANSI volume label.

User Action: Either write an ANSI label when initializing the tape or change the access in the program to device-specific.

147 ?RECORD number exceeds maximum

Explanation: ERROR—The specified record number exceeds the maximum specified for this file or the maximum record number was negative when the file was created.

User Action: Reduce the specified record number.

148 ?Bad RECORDSIZE value on OPEN

Explanation: ERROR—Either (1) the value in the RECORDSIZE clause is zero or greater than 16384, or (2) the value does not match the RECORDSIZE clause used when the file was created.

User Action: Change the value in the RECORDSIZE clause.

149 ?Not at end of file

Explanation: ERROR—The program attempted a PUT operation either on a sequential file before the last record, or without opening the file with an ACCESS WRITE clause.

User Action: Open a sequential file with an ACCESS APPEND clause or open the file with an ACCESS WRITE clause.

150 ?No primary key specified

Explanation: ERROR—The program attempts to create an INDEXED file without specifying a PRIMARY KEY value.

User Action: Specify a PRIMARY KEY value.

151 ?Key field beyond end of record

Explanation: ERROR—The position given for the key field exceeds the maximum size of the record.

User Action: Specify a key field within the record.

152 ?Illogical record accessing

Explanation: ERROR—The program attempts to perform an operation that is invalid for the specified file organization, for example, a random access on a sequential file.

User Action: Supply a valid operation for that file organization or change the file organization.

153 ?Record already exists

Explanation: ERROR—An attempted random access PUT operation on a RELATIVE file has encountered a pre-existing record.

User Action: Specify a different record number in the RECORD clause of the PUT statement or delete the existing record.

154 ?Record/bucket locked

Explanation: ERROR—The program attempts to access a record or bucket that has been locked by another program.

User Action: Try the operation again.

155 ?Record not found

Explanation: ERROR—A random access GET or FIND operation was attempted on a deleted or nonexistent record.

User Action: None.

156 ?Size of record invalid

Explanation: ERROR—The program contains a COUNT clause specification that is invalid because COUNT:

- Equals zero
- Exceeds the maximum size of the record
- Conflicts with the actual size of the current record during a sequential file UPDATE operation on disk
- Does not equal the maximum record size for fixed format records

User Action: Supply a valid COUNT value.

157 ?Record on file too big

Explanation: ERROR—The specified record is longer than the record buffer.

User Action: Increase the record buffer's size.

158 ?Primary key out of sequence

Explanation: ERROR—RMS–11 has detected an error in a sequential PUT operation to an INDEXED file.

User Action: Change the PUT statement. If this does not work, the file is corrupted and you cannot do anything.

159 ?Key larger than record

Explanation: ERROR—The key specification exceeds the maximum record size.

User Action: Reduce the size of the key specification.

160 ?File attributes not matched

Explanation: ERROR—The following attributes in the OPEN statement do not match the corresponding attributes of the target file:

- ORGANIZATION
- BUCKETSIZE
- BLOCKSIZE
- KEY
- Record format

User Action: Change the OPEN statement attributes to match those of the file or remove the clause.

161 ?Move overflows buffer

Explanation: ERROR—The combined length of elements in the MOVE statement exceeds the record size defined for the file.

User Action: Reduce the size of the elements in the MOVE statement or increase the file's record size.

162 ?Cannot open file

Explanation: ERROR—The specified file cannot be opened.

User Action: Check the STATUS variable for system error codes.

164 ?Terminal format file required

Explanation: ERROR—The program attempted to use PRINT #, INPUT #, LINPUT #, MAT INPUT #, MAT PRINT #, or PRINT USING # to access a RELATIVE, INDEXED, or VIRTUAL file.

User Action: Supply a terminal-format file.

165 ?Cannot position to EOF

Explanation: ERROR—The operating system could not find the end of a sequential file opened with ACCESS APPEND. The file could be corrupted.

User Action: None.

166 ?Negative fill or string length

Explanation: ERROR—A MOVE statement contains a FILL item or string length with a negative value.

User Action: Change the FILL item or string length value to be greater than or equal to zero.

167 ?Illegal record format

Explanation: ERROR—The record format is illegal for one of the following reasons:

- The specified record does not match the organization of the file.
- The specified record is illegal for the operating system on which the file resides.
- There are embedded carriage control characters in variable length records.

User Action: Correct the record format. Make sure the record has the same organization as specified when the file was created, that the record size is valid for the operating system, and that there are no embedded carriage control characters.

168 ?Illegal ALLOW clause

Explanation: ERROR—The value specified for the ALLOW clause is illegal for the type of file organization or for the operating system on which the file resides.

User Action: Change the ALLOW clause argument.

170 ?Index not fully optimized

Explanation: ERROR—A record was successfully written to an INDEXED file; however, the alternate key path was not optimized. This slows record access.

User Action: Delete the record and rewrite it.

171 ?RRV not fully updated

Explanation: ERROR—RMS–11 wrote a record successfully but did not update one or more Record Retrieval Vectors. Therefore, you cannot retrieve any records associated with those vectors.

User Action: Delete the record and rewrite it.

173 ?Invalid RFA field

Explanation: ERROR—During a FIND or GET operation by RFA, an invalid record's file address was contained in the RAB.

User Action: None. Please submit an SPR and include relevant output.

175 ?Bad node name

Explanation: ERROR—The specified node name is invalid, or, for NAME AS, the two node names are different.

User Action: Check node name.

180 ?No support for op in task

Explanation: FATAL—The program attempts to do the following:

- Open a file that has an organization that is not specified by a qualifier to the BUILD command.
- Perform an I/O operation that requires RMS–11 file support not included in the BUILD command.

User Action: Change the ORGANIZATION clause in the OPEN statement or use the BUILD command with the correct qualifier.

182 ?Network operation rejected

Explanation: ERROR—A DECnet operation has failed.

User Action: Check that the DECnet link is available. Check that the operation is supported both over the network and by the remote node.

183 ?REMAP overflows buffer

Explanation: ERROR—The combined length of elements in the REMAP statement exceeds the record buffer defined for the file.

User Action: Reduce the size of the elements in the REMAP statement or increase the size of the record buffer specified by the MAP statement.

184 ?Unaligned REMAP variable

Explanation: ERROR—A REMAP statement attempts to put a WORD variable on an odd byte boundary.

User Action: Change the variables in the REMAP statement to align on an even byte boundary.

185 %RECORDSIZE overflows MAP

Explanation: WARNING—The file's record size specified in the RECORDSIZE clause is larger than the storage allocated by the MAP statement.

User Action: Reduce the file's record size with the RECORDSIZE clause or increase the amount of storage allocated by the MAP statement.

186 ?Improper error handling

Explanation: ERROR—The program attempts to execute a RESUME statement in a program module other than the parent module or the module where the error occurred.

User Action: Change the program logic to execute a RESUME statement in the parent module or the module where the error occurs.

196 ?REMAP string not static

Explanation: ERROR—You referenced a string with a REMAP statement that was not declared in a COMMON or MAP statement.

User Action: Declare the string in a COMMON or MAP statement.

243 ?CHAIN to non-existent line no

Explanation: ERROR—RSTS/E only. The program attempts to chain to another program using a line number that does not exist in the target program.

User Action: Change the line number to an existing line number in the target program.

246 ?Error trap needs RESUME

Explanation: ERROR—An error handler attempts to execute an END without first executing a RESUME statement.

User Action: Change the program logic so that the error handler executes a RESUME statement before executing an END statement.

247 ?Illegal RESUME to subroutine

Explanation: ERROR—While in an error handler activated by a subroutine, the error handler attempts to RESUME without a line number.

User Action: None. You cannot use RESUME without a line number if the current module name does not match the error module name; that is, you cannot RESUME to a subroutine unless you specify a line number.

248 ?Illegal return from subroutine

Explanation: ERROR—An external subroutine tries to execute a RETURN statement before the CALL statement calling the subroutine is executed.

User Action: Change the program so that the CALL statement comes before the RETURN statement.

250 ?Not implemented

Explanation: ERROR—The program attempted to use a language feature that does not exist in this version of BASIC, for example, TIME(4%).

User Action: Do not use the feature.

251 ?Recursive subroutine call

Explanation: ERROR—The program contains a subroutine that attempts to call itself.

User Action: A subroutine cannot call itself. Correct the program logic.

252 ?File ACP failure

Explanation: ERROR—The operating system's file handler reported an error to RMS–11.

User Action: The corresponding error value is stored in the STATUS variable. See the *RSTS/E RMS–11 User's Guide* or the *RSX–11M/M–PLUS RMS–11 User's Guide* for more information.

253 ?Directive error

Explanation: ERROR—A system service call resulted in an error.

User Action: The corresponding error value is stored in the STATUS variable. See the *RSTS/E RMS–11 User's Guide* or the *RSX–11M/M–PLUS RMS–11 User's Guide* for more information.

B.4 Alphabetical List of Error Messages

Table B–1 contains an alphabetical list of the BASIC–PLUS–2 run-time errors and their associated error numbers. See the previous section for an explanation of these errors.

Table B–1 Alphabetical List of Run-Time Errors

Text	Number
?Account or device in use	3
?Arguments don't match	88
?Bad directory for device	1
?Bad node name	175
?Bad record identifier	143
?Bad RECORDSIZE value on OPEN	148
?Cannot open file	162
?Cannot open error file	None
?Cannot position to EOF	165
?Can't find file or account	5
?Can't invert matrix	56
?CHAIN to non-existent line no	243
?Corrupted file structure	29
%Data format error	50
?Device hung or write locked	14
?Device not available	8
?Device not file-structured	30
?Directive error	253
?Disk block is interlocked	19
?Disk error during swap	37
?Disk pack is locked out	22
?Disk pack is not mounted	21
?Disk pack is private	24
?Disk pack needs REBUILDing	25

(continued on next page)

Table B–1 (Cont.) Alphabetical List of Run-Time Errors

Text	Number
%Division by 0	61
?Duplicate key detected	134
?End of file on device	11
?Error trap needs RESUME	246
?Fatal disk pack mount error	26
?Fatal system I/O failure	12
?Field overflows buffer	63
?File ACP failure	252
?File attributes not matched	160
?File is locked	138
%Floating point error	48
?FNEND without function call	73
?I/O channel already open	7
?I/O channel not open	9
?I/O to detached keyboard	27
?Illegal ALLOW clause	168
%Illegal argument in log	53
?Illegal byte count for I/O	31
?Illegal cluster size	23
?Illegal file name	2
?Illegal I/O Channel	46
?Illegal key attributes	137
?Illegal MAGTAPE( ) usage	65
?Illegal number	52
?Illegal operation	141
?Illegal or illogical access	136
?Illegal record format	167
?Illegal record on file	142
?Illegal RESUME to subroutine	247

(continued on next page)

Table B–1 (Cont.) Alphabetical List of Run-Time Errors

Text	Number
?Illegal return from subroutine	248
?Illegal SYS( ) usage	18
?Illegal usage	135
?Illegal usage for device	133
?Illogical record accessing	152
%Imaginary square roots	54
?Improper error handling	186
?Index not fully optimized	170
?Index not initialized	140
%Integer error	51
?Integer overflow, FOR loop	60
?Invalid file options	139
?Invalid key of reference	144
?Invalid RFA field	173
?Key field beyond end of record	151
?Key larger than record	159
?Key not changeable	130
?Key size too large	145
?Keyboard wait exhausted	15
?Line too long	47
?Magtape record length error	40
?Matrix or array too big	44
?Maximum memory exceeded	126
?Memory management violation	35
?Memory parity failure	38
?Move overflows buffer	161
?Name or account now exists	16
?Negative fill or string length	166
?Network operation rejected	182

(continued on next page)

Table B–1 (Cont.) Alphabetical List of Run-Time Errors

Text	Number
?No buffer space available	32
?No current record	131
?No primary key specified	150
?No room for user on device	4
?No support for op in task	180
?Not a random access device	64
?Not a valid device	6
?Not at end of file	149
?Not enough data in record	59
?Not implemented	250
?Odd address trap	33
?ON statement out of range	58
?Out of data	57
?Pack IDs don't match	20
?Primary key out of sequence	158
?PRINT-USING format error	116
?Program lost-Sorry	103
?Programmable ^C trap	28
?Protection violation	10
?Record already exists	153
?Record has been deleted	132
?Record not found	155
?RECORD number exceeds maximum	147
?Record on file too big	157
?Record/bucket locked	154
%RECORDSIZE overflows MAP	185
?Recursive subroutine call	251
?Redimensioned array	105
?REMAP overflows buffer	183

(continued on next page)

Table B–1 (Cont.) Alphabetical List of Run-Time Errors

Text	Number
?Reserved instruction trap	34
?RESUME and no error	104
?RETURN without GOSUB	72
?RMS error # <num>	None
?RRV not fully updated	171
%SCALE factor interlock	127
?Size of record invalid	156
?SP Stack Overflow	36
?Subscript out of range	55
?Tape BOT detected	129
?Tape not ANSI labeled	146
?Tape records not ANSI	128
?Terminal format file required	164
?Too few arguments	97
?Too many arguments	89
?Too many open files on unit	17
?Unaligned REMAP variable	184
?User data error on device	13
?Virtual array not on disk	43
?Virtual array not yet open	45
?Virtual buffer too large	42

B.5 Non-BASIC Errors

The following errors are not generated by BASIC–PLUS–2; however, they can be displayed with the ERT$ function and are included for completeness.

Number	Text
39	?Magtape select error
41	?Non-res run-time system
49	%Argument too large in EXP
62	?No run-time system
66	?Missing special feature
67	?Illegal switch usage
68 – 70	Unused ERROR messages
71	?Statement not found
74	?Undefined function called
75	?Illegal symbol
76	?Illegal verb
77	?Illegal expression
78	?Illegal mode mixing
79	?Illegal IF statement
80	?Illegal conditional clause
81	?Illegal function name
82	?Illegal dummy variable
83	?Illegal FN redefinition
84	?Illegal line number(s)
85	?Modifier error
86	?Can't compile statement
87	?Expression too complicated
90	%Inconsistent function usage
91	?Illegal DEF nesting
92	?FOR without NEXT
93	?NEXT without FOR
94	?DEF without FNEND
95	?FNEND without DEF
96	?Literal string needed
98	?Syntax error
99	?String is needed

Number	Text
100	?Number is needed
101	?Data type error
102	?1 or 2 dimensions only
106	%Inconsistent subscript use
107	?ON statement needs GOTO
108	?End of statement not seen
109	?What
110	?Bad line number pair
111	?Not enough available memory
112	?Execute only file
113	?Please use the RUN command
114	?Can't CONTinue
115	?File exists-RENAME/REPLACE
117	?Matrix or array without DIM
118	?Bad number in PRINT-USING
119	?Illegal in immediate mode
120	?PRINT-USING buffer overflow
121	?Illegal statement
122	?Illegal FIELD variable
123	Stop
124	?Matrix dimension error
125	?Wrong math package
163	?No file name
169	?Unused ERROR message
172	?Record lock failed
174	?File expiration date unexpired
176 - 179	Unused ERROR messages
181	?Decimal overflow
187	?Illegal record locking clause
188 - 226	Unused ERROR messages
227	?String too long

Number	Text
228	?Record attributes not matched
229	?Differing use of /DOU
230	?No fields in image
231	?Illegal string image
232	?Null image
233	?Illegal numeric image
234	?Numeric image for string
235	?String image for numeric
236	?TIME limit exceeded
237	?1st arg to SEQ$ > 2nd
238	?Arrays must be same dimension
239	?Arrays must be square
240	?Cannot change array dimensions
241	?Floating overflow
242	?Floating underflow
244	?Exponentiation error
245	?Illegal exit from DEF
249	?Argument out of bounds
254 - 255	Unused ERROR messages

C

ASCII Codes and Data Representation

ASCII is a 7-bit character code with an optional parity bit (8) added for many devices. Programs normally use seven bits internally with the eighth bit being zero; the extra bit is either stripped (on input) or added by a device driver (on output) so the program will operate with either parity- or nonparity-generating devices. The eighth bit is reserved for future standardization.

The International Reference Version (IRV) of ISO Standard 646 is identical to the IRV in CCITT Recommendation V.3 (International Alphabet No. 5). The character sets are the same as ASCII except that the ASCII dollar sign (hexadecimal 24) is the international currency sign, which looks like ###.

ISO Standard 646 and CCITT V.3 also specify the structure for national character sets, of which ASCII is the U.S. national set. Certain specific characters are reserved for national use. These are the values and symbols:

Hexadecimal Value	IRV	ASCII
23	#	#
24	###	$ (General currency symbol vs. dollar sign)
40	@	@
5B	[	[
5C	\	\
5D	]	]
5E	^	^
60	`	`
7B	{	{
7C	|	|
7D	}	}
7E	( )	~ (Overline vs. tilde)

ISO Standard 646 and CCITT Recommendation V.3 (International Alphabet No. 5) are identical to ASCII except that the number sign (23) is represented as ## instead of #, and certain characters are reserved for national use.

Table C–1 ASCII Codes

Decimal Code	8-Bit Hexadecimal Code	Character	Remarks
0	00	NUL	Null (tape feed)
1	01	SOH	Start of heading (^A)
2	02	STX	Start of text (end of address, ^B)
3	03	ETX	End of text (^C)
4	04	EOT	End of transmission (shuts off the TWX machine ^D)
5	05	ENQ	Enquiry (WRU, ^E)
6	06	ACK	Acknowledge (RU, ^F)
7	07	BEL	Bell (^G)
8	08	BS	Backspace (^H)
9	09	HT	Horizontal tabulation (^I)
10	0A	LF	Line feed (^J)
11	0B	VT	Vertical tabulation (^K)
12	0C	FF	Form feed (page, ^L)
13	0D	CR	Carriage return (^M)
14	0E	SO	Shift out (^N)
15	0F	SI	Shift in (^O)
16	10	DLE	Data link escape (^P)
17	11	DC1	Device control 1 (^Q)
18	12	DC2	Device control 2 (^R)
19	13	DC3	Device control 3 (^S)
20	14	DC4	Device control 4 (^T)
21	15	NAK	Negative acknowledge (ERR, ^U)
22	16	SYN	Synchronous idle (^V)
23	17	ETB	End-of-transmission block (^W)

(continued on next page)

Table C–1 (Cont.) ASCII Codes

Decimal Code	8-Bit Hexadecimal Code	Character	Remarks
24	18	CAN	Cancel (^X)
25	19	EM	End of medium (^Y)
26	1A	SUB	Substitute (^Z)
27	1B	ESC	Escape (prefix of escape sequence)
28	1C	FS	File separator
29	1D	GS	Group separator
30	1E	RS	Record separator
31	1F	US	Unit separator
32	20	SP	Space
33	21	!	Exclamation point
34	22	"	Double quotation mark
35	23	#	Number sign
36	24	$	Dollar sign
37	25	%	Percent sign
38	26	&	Ampersand
39	27	'	Apostrophe
40	28	(	Left (open) parenthesis
41	29	)	Right (close) parenthesis
42	2A	*	Asterisk
43	2B	+	Plus sign
44	2C	,	Comma
45	2D	–	Minus sign, hyphen
46	2E	.	Period (decimal point)
47	2F	/	Slash (slant)
48	30	0	Zero
49	31	1	One
50	32	2	Two

(continued on next page)

Table C–1 (Cont.) ASCII Codes

Decimal Code	8-Bit Hexadecimal Code	Character	Remarks
51	33	3	Three
52	34	4	Four
53	35	5	Five
54	36	6	Six
55	37	7	Seven
56	38	8	Eight
57	39	9	Nine
58	3A	:	Colon
59	3B	;	Semicolon
60	3C	<	Less than (left angle bracket)
61	3D	=	Equal sign
62	3E	>	Greater than (right angle bracket)
63	3F	?	Question mark
64	40	@	Commercial at
65	41	A	Uppercase A
66	42	B	Uppercase B
67	43	C	Uppercase C
68	44	D	Uppercase D
69	45	E	Uppercase E
70	46	F	Uppercase F
71	47	G	Uppercase G
72	48	H	Uppercase H
73	49	I	Uppercase I
74	4A	J	Uppercase J
75	4B	K	Uppercase K
76	4C	L	Uppercase L
77	4D	M	Uppercase M

(continued on next page)

Table C–1 (Cont.) ASCII Codes

Decimal Code	8-Bit Hexadecimal Code	Character	Remarks
78	4E	N	Uppercase N
79	4F	O	Uppercase O
80	50	P	Uppercase P
81	51	Q	Uppercase Q
82	52	R	Uppercase R
83	53	S	Uppercase S
84	54	T	Uppercase T
85	55	U	Uppercase U
86	56	V	Uppercase V
87	57	W	Uppercase W
88	58	X	Uppercase X
89	59	Y	Uppercase Y
90	5A	Z	Uppercase Z
91	5B	[	Left square bracket
92	5C	\	Backslash (reverse slant)
93	5D	]	Right square bracket
94	5E	^	Circumflex (caret)
95	5F	_	Underscore (underline)
96	60	`	Grave accent
97	61	a	Lowercase a
98	62	b	Lowercase b
99	63	c	Lowercase c
100	64	d	Lowercase d
101	65	e	Lowercase e
102	66	f	Lowercase f
103	67	g	Lowercase g
104	68	h	Lowercase h

(continued on next page)

Table C–1 (Cont.) ASCII Codes

Decimal Code	8-Bit Hexadecimal Code	Character	Remarks
105	69	i	Lowercase i
106	6A	j	Lowercase j
107	6B	k	Lowercase k
108	6C	l	Lowercase l
109	6D	m	Lowercase m
110	6E	n	Lowercase n
111	6F	o	Lowercase o
112	70	p	Lowercase p
113	71	q	Lowercase q
114	72	r	Lowercase r
115	73	s	Lowercase s
116	74	t	Lowercase t
117	75	u	Lowercase u
118	76	v	Lowercase v
119	77	w	Lowercase w
120	78	x	Lowercase x
121	79	y	Lowercase y
122	7A	z	Lowercase z
123	7B	{	Left brace
124	7C	|	Vertical line
125	7D	}	Right brace
126	7E	~	Tilde
127	7F	DEL	Delete (rubout)

C.1 Radix–50 Character Set

Many items, such as file names and file types, are stored in Radix–50 format. This format allows three characters of data to be stored as a 2-byte integer (one 16-bit word).

Table C–2 lists the characters representable in Radix–50 format, together with their ASCII octal and Radix–50 octal equivalents.

Table C–2 Radix–50 Character Set

ASCII Character	ASCII Octal Equivalent	Radix–50 Octal Equivalent
space	40	0
A through Z	101 through 132	1 through 32
$	44	33
.	56	34
0 through 9	60 through 71	36 through 47

Radix–50 evaluates a character according to the format:

X = Y * 50^Z

X Is the value of the character.

Y Is the Radix–50 octal equivalent of the character.

50 Is a constant (in octal).

Z Is the character's position in the string. The leftmost digit is assigned position two, the middle character is assigned position one, and the rightmost character is assigned position zero.

To represent a 3-character alphanumeric string in Radix–50 format, the first character is placed in the leftmost position of the Radix–50 word. For example, in the string X2B, the character X (30 octal) is multiplied by 50^2 to give 113000 (octal). The character 2 (40 octal) is multiplied by 50^1 to give 002400. The character B (2 octal) is multiplied by 50^0 to give 000002. Adding the value of each character gives the full octal value of the Radix–50 word.

$$X = 30 * 50^2 = 113000$$
$$2 = 40 * 50^1 = 002400$$
$$B = 02 * 50^0 = 000002$$
TOTAL = 115402 (octal)

Note that addition is also carried out in octal.

Table C–3 simplifies this process by listing the value of each Radix–50 character for each position.

Table C–3 ASCII and Radix–50 Equivalents

First or Single Character		Second Character		Third Character	
space	000000	space	000000	space	000000
A	003100	A	000050	A	000001
B	006200	B	000120	B	000002
C	011300	C	000170	C	000003
D	014400	D	000240	D	000004
E	017500	E	000310	E	000005
F	022600	F	000360	F	000006
G	025700	G	000430	G	000007
H	031000	H	000500	H	000010
I	034100	I	000550	I	000011
J	037200	J	000620	J	000012
K	042300	K	000670	K	000013
L	045400	L	000740	L	000014
M	050500	M	001010	M	000015
N	053600	N	001060	N	000016
O	056700	O	001130	O	000017
P	062000	P	001200	P	000020
Q	065100	Q	001250	Q	000021
R	070200	R	001320	R	000022
S	073300	S	001370	S	000023
T	076400	T	001440	T	000024
U	101500	U	001510	U	000025
V	104600	V	001560	V	000026
W	107700	W	001630	W	000027
X	113000	X	001700	X	000030
Y	116100	Y	001750	Y	000031
Z	121200	Z	002020	Z	000032
$	124300	$	002070	$	000033
.	127400	.	002140	.	000034

(continued on next page)

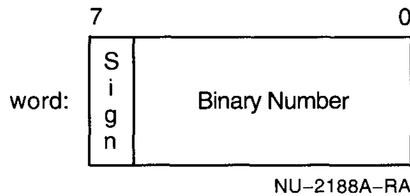
Table C–3 (Cont.) ASCII and Radix–50 Equivalents

First or Single Character		Second Character		Third Character	
0	135600	0	002260	0	000036
1	140700	1	002330	1	000037
2	144000	2	002400	2	000040
3	147100	3	002450	3	000041
4	152200	4	002520	4	000042
5	155300	5	002570	5	000043
6	160400	6	002640	6	000044
7	163500	7	002710	7	000045
8	166600	8	002760	8	000046
9	171700	9	003030	9	000047

C.2 BYTE Integer Format

Figure C–1 shows the format of a BYTE integer value.

Figure C–1 Byte-Length Integer Format



BYTE integers are stored in sign-extended two's complement representation. For example, here are the octal values for four different binary numbers:

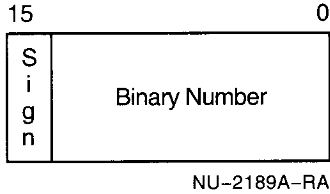
decimal	octal
+6	= 006
+22	= 026
–7	= 371
–1	= 377

BYTE integer constants must be in the range –128 to +127.

C.3 WORD Integer Format

Figure C–2 shows the format of a WORD integer value.

Figure C–2 Word-Length Integer Format



WORD integers are stored in sign-extended two's complement representation. For example, here are the octal values for four different binary numbers:

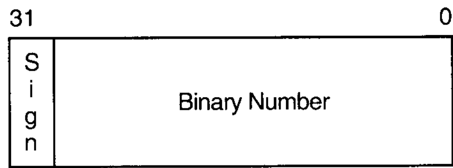
decimal	octal
+6	= 000006
+22	= 000026
–7	= 177771
–1	= 177777

WORD integer constants must be in the range –32768 TO +32767.

C.4 LONGWORD Integer Format

Figure C–3 shows the format of a LONG integer value.

Figure C–3 Longword Integer Format



LONG integers are stored in sign-extended two's complement representation. For example, here are the octal values for four different binary numbers:

decimal	octal
+6	= 000000000006
+22	= 000000000026
–7	= 37777777771
–1	= 37777777777

LONG integer constants must be in the range –2147483648 to +2147483647.

C.5 Floating-Point Formats

The exponent for both 2-word and 4-word floating-point formats is stored in excess 128 (200 octal) notation. Binary exponents from –128 to +127 are represented by the binary equivalents of zero through 255 (zero through 377 octal).

Fractions are represented in sign-magnitude notation, with the binary radix point to the left.

Numbers are assumed to be normalized. The most significant bit is assumed to be 1 and is not stored. However, if the exponent is 0, the bit is also 0. The value zero is represented by two or four words of zeros. Figure C–4 shows the format of floating-point values.

Figure C–4 Floating-Point Format

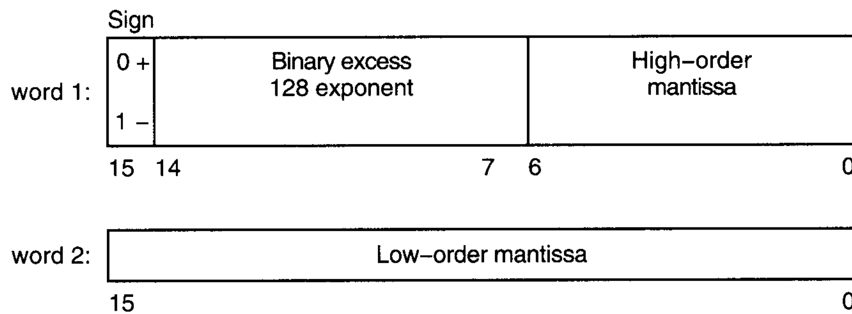
Number	2–word Format	4–word Format
+1.0	40200 0	40200 0 0 0 0
-5	140640 0	140640 0 0 0 0

NU–2191A–RA

C.5.1 Single-Precision Format

Two words describe each single-precision floating-point value. Figure C–5 shows the format of the description.

Figure C–5 Single-Precision Format

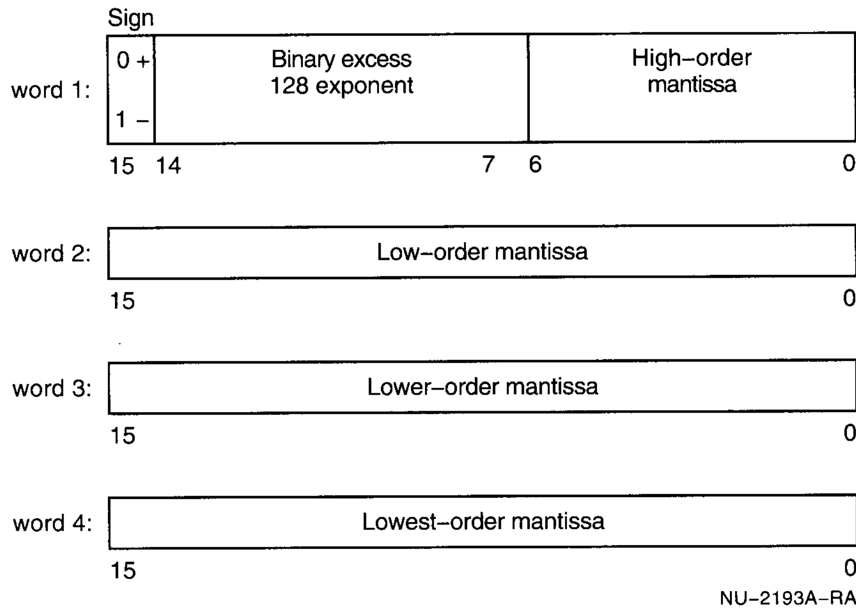


The effective precision is 23 bits (six digits of accuracy) and the magnitude range is .29E–38 to .17E39.

C.5.2 Double-Precision Format

Four words describe each double-precision floating-point value. Figure C–6 shows the format of the description.

Figure C–6 Double-Precision Format

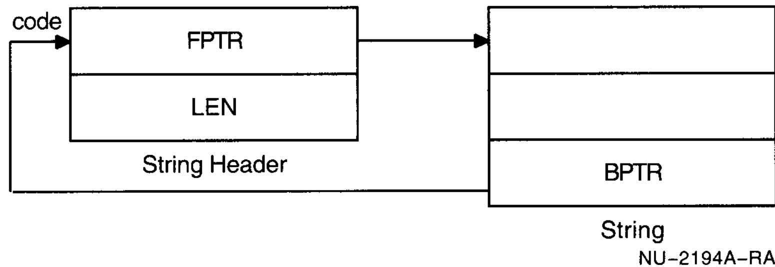


The effective precision is 55 bits (16 decimal digits of accuracy) and the magnitude range is .29E–38 to .17E39.

C.6 String and Array Formats

Figure C–7 shows the format of a dynamic string. Each box represents a word.

Figure C–7 Dynamic String Format



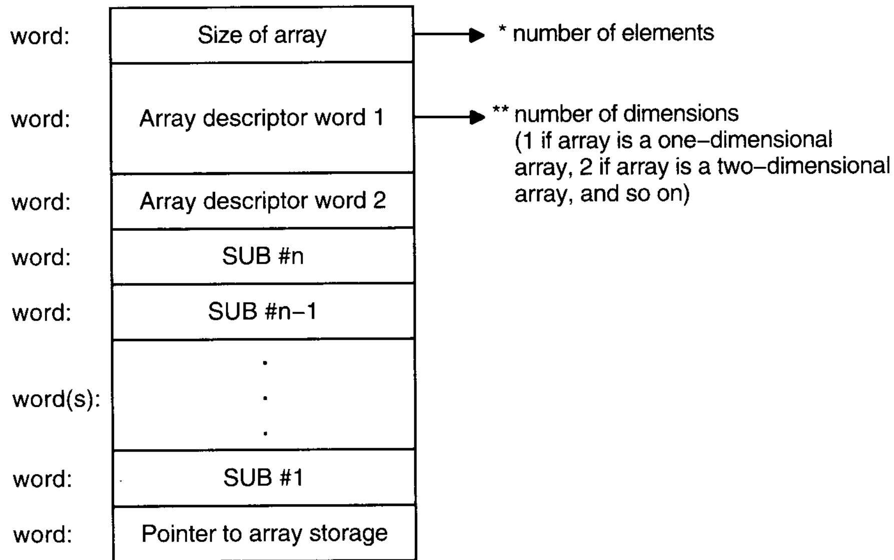
Dynamic strings contain a 2-word string header. The first word is a forward pointer (FPTR) that points to the first character of the string. The second word represents the length (LEN) of the string in bytes. Following the data in the string and aligned on the next higher word boundary is a word that points back to the forward pointer. This word is internally specific and should not be accessed.

C.6.1 Array Formats

Arrays in memory contain two array descriptor words. The first descriptor word defines the number of dimensions, as described in the following figures. The second array descriptor word is described in C.6.2.

Figure C–8 shows the format of arrays in memory.

Figure C–8 Format of Arrays in Memory



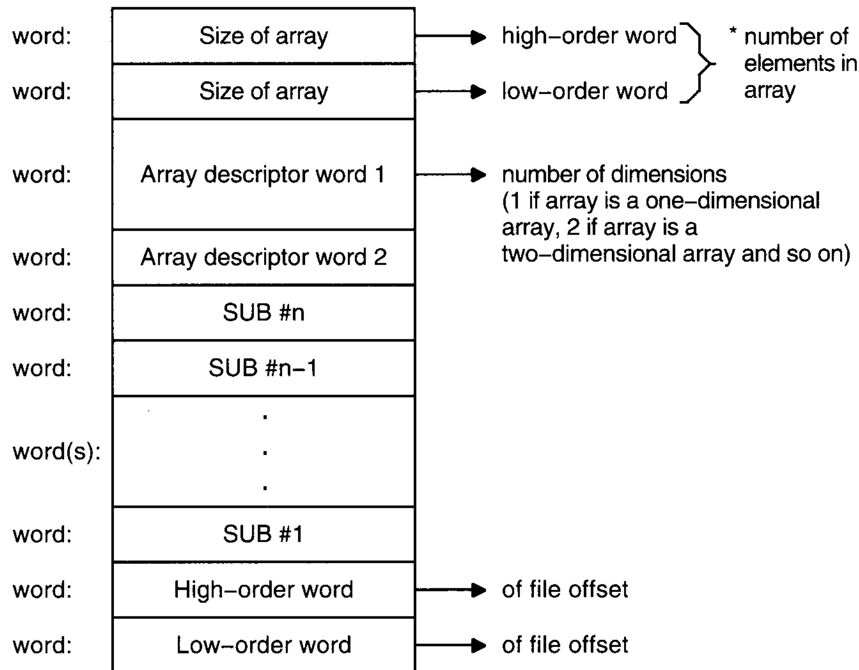
*   The first word contains the number of elements in the array if the array is redimensioned by a MATRIX statement or the array is used as a subroutine argument. Otherwise, the word describing the size of an array does not exist.

**   If the array appears in a MAP DYNAMIC statement, the value of array descriptor 1 is 256 plus the number of dimensions.

If a MACRO subprogram accepts an array passed by a descriptor as an argument, the array pointer in the argument list will point to SUB *n*.

Figure C–9 shows the format of virtual arrays.

Figure C–9 Format of Arrays in Virtual Memory

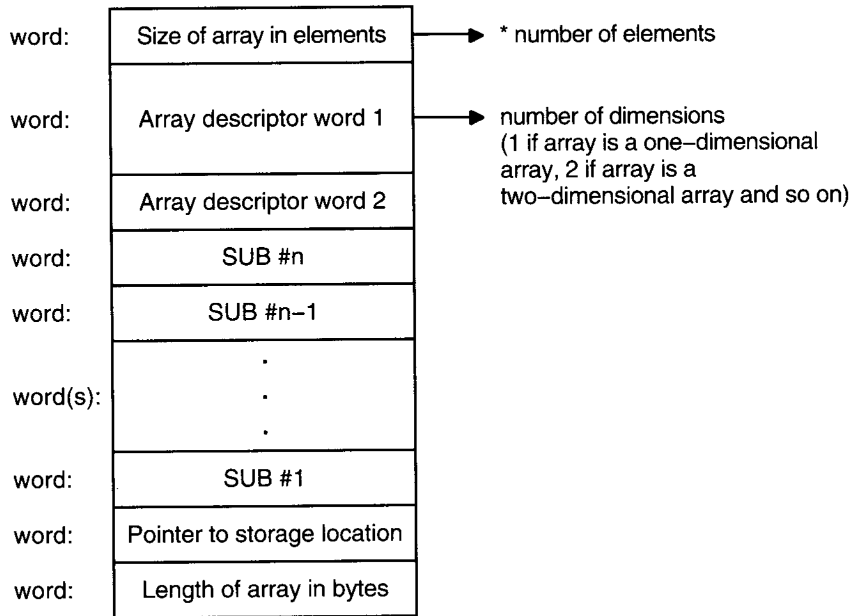


* The high- and low-order words point to the number of elements in the array if the array is redimensioned by a MATRIX statement or if the array is used as a subroutine argument; otherwise, the word describing the size of an array does not exist.

If a MACRO subprogram accepts an array passed by a descriptor as an argument, the array pointer in the argument list will point to SUB #n.

Figure C–10 shows the format of dynamic arrays.

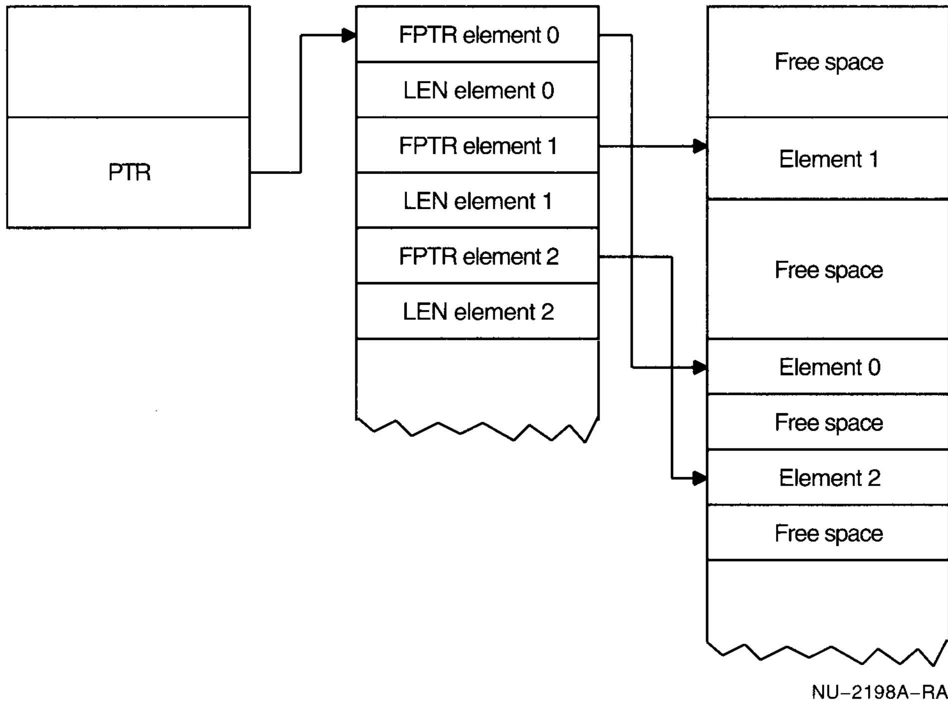
Figure C–10 Dynamic Arrays



* The first word contains the number of elements in the array if the array is redimensioned by a MATRIX statement or the array is used as a subroutine argument; otherwise, the word describing the size of an array does not exist.

Figure C–11 shows the format of dynamic string array pointers.

Figure C–11 Dynamic String Array Pointers



C.6.2 Array Descriptor Word 2

The second array descriptor word is a 16-bit word used by BASIC to describe the characteristics of an array. The bits of array descriptor word 2 are explained in Figure C–12.

Figure C–12 Array Descriptor Word 2

Array Type	Bits															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Numeric Memory	0	L	0	T			0	0	0	0	0	0	0	0	0	0
Numeric Virtual	0	0	1	T			0	0	Channel number							
String Memory	1	0	0	T			0	0	0	0	0	0	0	0	0	0
String Common	1	1	0	0	Element length in bytes											
String Virtual	1	0	1	LOG2(LEN)					Channel number							

NU–2199A–RA

Bit 12 no longer contains dimension information. Instead, along with bits 11 and 10, it contains type information.

Note that although the STRING and LONGWORD data types have the same code, string arrays have bit 15 set.

T-Data Type

- BYTE = 000
- WORD = 001
- LONG = 010
- SINGLE = 110
- DOUBLE = 111
- STRING = 010
- RFA = 100

L - Location (memory or common)

Each array sets the bits of the array descriptor word as follows:

Numeric memory	Bits 0 through 9 are set to 0. Bits 10 through 12 set the data type (for example, 001 for word, 110 for single-precision, 111 for double-precision). Bit 13 is set to 0. Bit 14 is set to 0 if the array is in memory and 1 if the array is a COMMON. Bit 15 is set to 0.
Numeric virtual	Bits 0 through 7 represent the channel number. Bits 8 and 9 are set to 0. Bits 10 through 12 set the data type. Bit 13 is set to 1. Bits 14 and 15 are set to 0.

String memory	Bits 0 through 9 are set to 0. Bits 10 through 12 set the data type. Bits 13 and 14 are set to 0. Bit 15 is set to 1.
String common	Bits 0 through 11 represent the element length in bytes. Bits 12 and 13 are set to 0. Bits 14 and 15 are set to 1.
String virtual	Bits 0 through 7 represent the channel number. Bits 8 through 12 represent LOG2 of the string length. Bit 13 is set to 1. Bit 14 is set to 0. Bit 15 is set to 1.

The maximum number of elements is present in the array descriptor only when the array is redimensioned by a MATRIX statement or when the array is used as a subroutine argument.

Index

A

%ABORT directive, 14–9
ABS function, 8–2
ACCESS clause, 17–8
Addition
  array, 10–20
Ampersand (&)
  data, 5–6
  in comment field, 4–5
  MAT INPUT statement, 10–14
Ampersand (&), 1–3
APPEND command, 1–13
Arithmetic
  exectution time, 20–4
Array, 4–9, 10–1 to 10–23
  accessing, 12–14
  addition, 10–20
  arithmetic, 10–20
  assigning data to, 10–12
  assigning values to, 10–10, 10–13
  bounds, 4–10
  column, 10–16
  computation, 10–19
  creating, 10–2 to 10–8
  definition of, 4–10
  descriptor format, C–18
  determinant, 10–23
  dimensions, 10–9
  displaying, 10–9
  elements, 4–10, 10–7, 10–10, 10–13
  explicit, 10–2
  filling, 10–13
  fixed-length string, 10–6
Array (cont'd)
  format, 10–16, C–14
  I/O functions, 10–17
  implicit, 10–7
  input, 10–18 to 10–19
  inverting, 10–22
  list, 10–1
  matrix, 10–1
  matrix functions, 10–20
  multiplication, 10–20
  one-dimensional, 10–1
  output, 10–18 to 10–19
  overlaying, 10–7
  passing, 11–12
  printing elements, 10–16
  reassigning values, 10–20
  record buffer, 10–7
  redimensioning, 10–5, 10–15, 11–12
  referencing, 10–3, 10–7
  row, 10–16
  sharing, 10–6
  string elements, 10–15
  subscripted variable, 4–10
  subscripts, 10–1 to 10–2
  subtraction, 10–20
  transposing, 10–22
  two-dimensional, 10–1, 10–17
  types of, 4–10
  undeclared element, 10–7
  values, 10–9
  vector, 10–1
  virtual, 9–1, 12–5, 12–14
  zero-based, 10–1
ASCII
  character codes, C–1 to C–6

ASCII (cont'd)
  character set, 4–4
ASCII function, 8–7
Asterisk (*)
  PRINT USING statement, 13–8

B

BASIC character set, 4–4
BASIC command
  /CROSS_REFERENCE qualifier, 2–7, 2–9
  /DEBUG qualifier, 3–1
  description of, 2–2
  format of, 2–2
  /LIST qualifier, 2–7
  /[NO]BOUND qualifier, 2–4
  /[NO]BUILD qualifier, 2–4
  /[NO]/BUILD qualifier, 2–12
  /[NO]CHAIN qualfier, 2–4
  /[NO]CROSS_REFERENCE qualifier, 2–4
  /[NO]DEBUG qualfier, 2–5
  /[NO]FLAG qualifier, 2–5
  /[NO]LINES qualifier, 2–5
  /[NO]LIST qualifier, 2–5
  /[NO]MACRO qualifier, 2–6
  /[NO]OBJECT qualifier, 2–6
  /[NO]WARNINGS qualifier, 2–7
  qualifiers, 2–2 to 2–7
  /SCALE qualifier, 2–6
  /USING qualifier, 2–6
  /VARIANT qualifier, 2–6
Binary dump, 19–12
Block
  CASE, 6–13
  data, 12–1
  decision, 6–1
  descriptor, 11–31
  IF...THEN...ELSE, 4–3
  loop, 6–3
  RMS, 12–1
  SELECT, 6–13
Bounds
  See Array
Branch, 11–20
Breakpoint
  setting, 3–1, 3–3
BRLRES command, 1–13
Bucketsize, 12–37
  default value, 12–38
Buffer
  I/O, 12–6
  record, 12–6
BUILD command, 1–13, 2–12
  /CLUSTER qualifier, 18–11
  CMD file, 1–13
  /IDS qualifier, 16–1
  memory-resident library, 1–17
  ODL file, 1–13
  subprogram, 11–18
Byte
  boundary, 11–16
  format, C–9
BYTE data type, 4–6

C

CALL statement, 11–2, 11–5
  BY clause, 11–25
  format, 11–6
  parameter, 11–6
  subprogram, 11–6
CASE block, 6–13
Character
  nonprinting, 4–4
Character set
  ASCII, 4–4, C–1
  BASIC, 4–4
  RAD-50, C–6
CHR$ function, 8–7
CLOSE statement, 12–27
  ending I/O to a tape, 17–23
CMD file, 2–2
  generating, 2–12
Column, 10–16
Comma (,)
  in PRINT statement, 5–9
  MAT PRINT statement, 10–16
  PRINT USING statement, 13–7

Comma (,)
EXTRACT command, 1–16
Command file
See CMD file
Comment
in environment, 1–8
in programs, 4–4
Comment field, 4–4
data, 5–6
Common, 11–12
array, 10–6
block, 11–13
data type, 11–13
defining area, 7–15
initializing, 11–41
name, 11–13
numeric variable, 11–16
PSECT, 11–16
storage, 11–15
COMMON statement, 7–9, 10–6
example of, 11–14
format, 11–13
with subprograms, 7–14
Communication
task-to-task, 17–42
Compilation
aborting, 14–8, 14–9
conditional, 14–10
controlling, 14–8
displaying message during, 14–10
errors, A–1 to A–53
Compilation listing
altering, 14–2
controlling, 14–5
cross-reference section, 2–7, 2–9, 14–6
default file type, 2–7
paging, 14–4
qualifier summary section, 2–11
source program section, 2–7
subtitle, 14–3
title, 14–2
version number, 14–4
COMPILE command, 1–10, 1–14
Compile-time error
list of, A–1 to A–53
Compiler
See also Environment
functions of, 2–2
invoking, 1–2
invoking from DCL, 2–2
options, 1–14
Compiler directive, 14–1 to 14–11
compilation listing, 14–2
definition of, 14–1
rules of use, 14–1
uses of, 14–1
Conditional expressions
in IF...THEN...ELSE statement, 6–10
in WHILE...NEXT loops, 6–7
Constant
changing value of, 14–8
data type, 7–7
declaring, 7–6
definition of, 7–6
external, 7–7
lexical, 14–8, 14–9
named, 7–6
string, 9–1
Control statement, 6–1 to 6–21
execution time of, 20–4
COS function, 8–3
%CROSS directive, 14–6
Cross-reference listing, 2–9, 14–6
CTRLC function, 8–16, 15–7
Ctrl/C trapping, 8–16, 15–7
Currency symbol
PRINT USING statement, 13–9

D

Data
block, 12–1
format error, 15–3
formatting, 13–1
record, 12–1
representation, C–9 to C–20
rereading, 5–7
sharing, 5–7, 11–12
string, 9–5

Data format
  array, C–14
  byte integer, C–9
  double-precision, C–13
  floating-point, C–11
  longword integer, C–10
  single-precision, C–12
  string, C–14
  word integer, C–10
DATA statement, 5–6 to 5–7
  arrays, 10–12
  comments in, 5–6
  continuing, 5–6
Data type, 7–1 to 7–19
  assigning, 7–1
  BYTE, 4–6
  constant, 7–7
  declaring, 7–5
  definition of, 7–1
  explicit, 7–2
  explict, 4–5
  floating-point, 4–6, 7–2
  function, 8–2
  implicit, 4–5, 7–2
  integer, 7–1, 7–3
  INTEGER, 4–6
  keywords, 7–14
  LONGWORD, 4–6
  promotions, 7–8
  REAL, 4–6
  RFA, 4–6, 7–2
  selecting, 20–3
  size, 7–4
  string, 7–1
  STRING, 4–6
  subprogram, 11–5
  subtype, 7–4
  suffix, 7–2
  types of, 4–6
  WORD, 4–6
DATE$ function, 8–14
DCL command
  BASIC, 2–2
  LINK, 2–13
  RUN, 2–14
DCL command (cont'd)
  TKB, 2–12
DCL commands
  LIBRARY, 18–5
DCL level
  compiling programs at, 2–2
  developing programs at, 2–1
  linking programs at, 2–13
  running programs at, 2–14
  using debugger at, 3–7
Debugger, 3–1 to 3–20
  commands, 3–3
  error messages, 3–18
  invoking, 3–1
  sample session, 3–4
Decision block, 6–1
Decision structure, 6–10
Declaration, 7–1 to 7–19
Declarative statement, 7–1
DECLARE statement, 7–5, 10–3 to 10–4
Declining features, 2–5
DECnet
  file access, 18–12
  I/O, 17–41
DEF function
  control, 8–23
  error handler, 15–9
  multi-line, 8–19
  parameters, 8–23
  recursion, 8–20
  single-line, 8–18
DEF statement, 8–17, 11–2
Default
  displaying environment, 1–23
  error handler, 15–2
  initialization file, 1–26
  program file type, 2–3
  setting environment, 1–23
DELETE command, 1–15
DELETE command>with comma (,), 1–15
DELETE command>with hyphen (-), 1–15
DELETE statement, 12–20 to 12–21
Descriptor block, 11–31
DET function, 10–23

Device-specific I/O
  to disks, 17–34
  to tape, 17–19
  to unit record devices, 17–19
DIF$ function, 8–11
DIMENSION statement, 10–4 to 10–6
  array dimensioning, 10–5
  declarative, 10–5
  executable, 10–5
Disk unit
  allocating, 17–35
Disks
  accessing, 17–34
  creating, 17–36
  opening, 17–36
Documentation
  online, 1–17
Double-precision
  format, C–13
DSKLIB command, 1–15, 18–5, 18–6
Dump Analyzer Utility, 19–12
  dump file, 19–13
Dynamic mapping, 7–17, 12–7 to 12–9
Dynamic storage, 4–7
  allocating, 7–9
Dynamic string, 9–2

E

ECHO function, 8–17
EDIT command
  editing mode, 1–15
  /EDT qualifier, 2–1
  line mode, 1–15
  /RECOVER qualifier, 2–1
EDIT$ function, 9–16
Editing
  command mode, 1–16
  environment, 1–15
  journal file, 2–1
  line mode, 1–16
  program, 1–4
  prompt, 2–1
EDT editor, 2–1
  online help, 2–1
ELSE clause, 6–10
END IF statement, 6–11
END statement, 6–19, 6–21
Environment, 1–1 to 1–26
  appending programs in, 1–13
  building programs in, 1–13
  CMD file, 1–13
  compiling programs in, 1–4, 1–10, 1–14
  continuing lines in, 1–3
  creating programs in, 1–3 to 1–4
  DCL commands in, 1–12
  debugging in, 3–4
  default memory-resident library, 1–13, 1–17
  default object module library, 1–15
  default ODL file, 1–19
  defaults, 1–10
  deleting program lines in, 1–15
  developing programs in, 1–1 to 1–27
  displaying program in, 1–18
  editing in, 1–15
  entering, 1–1
  exiting from, 1–7, 1–16
  identification message, 1–2, 1–17
  invoking, 1–1
  line numbers in, 1–2
  linking programs in, 1–4
  listing file, 1–14
  loading object modules in, 1–18
  locating programs in, 1–20
  multi-unit programs in, 1–6
  naming programs in, 1–19
  object modules in, 1–10
  ODL file, 1–13
  options, 1–14
  program lines in, 1–2
  prompt, 1–2
  renaming programs in, 1–20
  replacing programs in, 1–20
  running programs in, 1–4, 1–20
  running subprograms in, 1–6
  statements in, 1–2
  Task Builder, 1–13
Environment commands, 1–10 to 1–26

ERL function, 15–5
ERN$ function, 15–6
ERR function, 15–4
Error
  analyzing fatal, 19–13
  anticipating, 15–2
  clearing, 15–11
  common, 15–3
  compile-time, A–1 to A–53
  correcting, 15–12
  Ctrl/C, 15–7
  diagnosing, 15–2
  end-of-file, 15–10
  environment, A–1 to A–53
  fatal, 15–1, 15–2, A–1, B–1
  format, A–1, B–2
  identifying, 15–4
  information, 15–1, 15–4
  level, A–1
  line number, 15–5
  message text, 15–6
  non-BASIC, 15–4
  number, 15–3, 15–4, B–1
  online information about, 1–17
  Optimizer Utility, 19–9
  output, 13–17
  pending, 15–8
  print, 13–17
  program name, 15–6
  Resequencer Utility, 19–16
  run-time, 15–1, B–1 to B–32
  severity, 15–1, A–1, B–1
  subprogram, 15–6, 15–8
  testing, 15–3
  text, A–1, B–1
  trappable, 15–1
  trapping, 15–1
  warning, 15–1
Error handler, 15–1 to 15–12
  control, 15–10
  Ctrl/C, 15–7
  CTRLC function, 15–7
  default, 15–2, 15–9, 15–10
  definition of, 15–1
  ERL function, 15–5
Error handler (cont'd)
  ERN$ function, 15–6
  ERR function, 15–4
  error condition, 15–12
  error number, 15–4
  ERT$ function, 15–6
  falling through, 15–12
  functions, 15–4 to 15–8
  intermittent, 15–10
  leaving, 15–11
  line number, 15–5
  loop variables, 15–12
  message text, 15–6
  program name, 15–6
  resuming compilation, 15–12
  subprogram, 15–6, 15–8
  types of, 15–1
  user-written, 15–2
ERT$ function, 15–6
Exception
  See Error
Exclamation point (!), 4–4
Execution
  of large tasks, 19–1
  resuming, 15–11
  stopping, 6–19
  suspending, 6–19
EXIT command, 1–16
EXIT statement, 6–15 to 6–16
EXP function, 8–5
Exponential format
  with asterisk fill, 13–11
Expression, 4–12
  conditional, 14–10
Expressions
  mixed-mode, 7–7
EXTERNAL statement, 7–7, 11–5
  data type, 11–5
  definition, 11–5
  format, 11–5
  parameter, 11–5
EXTRACT command, 1–16
EXTTSK option, 20–9

F

Fatal error, 15–2, A–1
Field, 12–1
  blank-if-zero, 13–12
  centered, 13–15
  E format, 13–10
  extended, 13–15
  leading zeros, 13–11
  left-justified, 13–14
  negative, 13–10
  record buffer, 12–9
  right-justified, 13–14
  zero-fill, 13–11
File
  alternate keys, 12–4
  closing, 12–27 to 12–28
  CMD, 2–2
  compiling from DCL, 2–2
  DECnet access to, 18–12
  deleting, 12–28
  determining organization, 12–28
  editing, 2–1
  error handling, 15–2
  file-related functions, 12–28 to 12–37
  files-11 format, 12–1
  I/O, 12–1 to 12–46, 17–1
  including external, 14–7
  indexed, 12–4
  line numbers, 14–7
  native mode, 12–1, 12–3
  object module, 2–13
  ODL, 2–2
  opening, 12–12
  operations, 12–11
  organization, 12–1, 12–3
  primary keys, 12–4
  random access, 12–4
  relative, 11–18, 12–4
  renaming, 12–27
  restoring, 12–26
  returning file name, 12–29
  returning status, 12–29
  RMS, 12–1, 17–1
File (cont'd)
  sequential, 12–3
  shared access, 12–4
  status of, 12–37
  structure, 17–18
  tape, 17–2
  task image, 2–12
  terminal-format, 5–14, 12–3
  transferring data to, 12–26
  truncating, 12–26
  virtual array, 12–5, 12–14
File name
  specifying in the OPEN statement, 17–1
File type
  BASIC program, 2–3
  listing file, 2–7
  object module, 2–13
  task image, 2–12
Files–11 file, 12–1
FILL
  formats, 7–13
  items, 7–13
FIND statement, 12–15 to 12–16
  random access, 12–15
  record pointer, 12–15
  sequential, 12–15
FIX function, 8–2
Fixed-length record, 12–2
Fixed-length string, 9–1
Floating-point
  data type, 7–3
  format, C–11
  numbers, 13–1
  PRINT statement, 13–1
  variables, 4–9
FOR modifier, 6–1
FOR statement
  in immediate mode statements, 1–9
FOR...NEXT loops, 6–4 to 6–7
Format
  ANSI, 17–2
  array, C–14
  byte-length integer, C–9
  centered, 13–15
  double-precision, C–13

Format (cont'd)
dynamic string, C–14
exponential, 13–10
fields, 13–2
floating-point, C–11
integer, 13–3
left-justified, 13–14
longword integer, C–10
negative field, 13–10
output, 5–9, 13–1
right-justified, 13–14
run-time error, B–2
single-precision, C–12
special symbols, 13–6
strings, 13–2
undefined, 17–18
word-length integer, C–10
FORMAT$ function, 8–8
FREE statement, 12–24
FSP$ function, 12–28
FSS$ function, 12–29
Function, 8–1 to 8–26
ASCII, 8–7
built-in, 8–1 to 8–17, 14–9, 15–4
Ctrl/C trapping, 15–7
data conversion, 8–7
data type, 7–5, 8–2
date and time, 8–14 to 8–16
declaring, 8–21
DEF, 8–17
definition of, 8–1
error handling, 15–4 to 15–8
external, 8–1, 8–24
file-related, 12–28 to 12–37
lexical, 14–9
multi-line, 8–19
naming, 8–17, 8–19, 8–21
numeric, 8–2
numeric string, 8–8
parameter list, 8–18
recursion in, 8–22
string, 9–9 to 9–17
string arithmetic, 8–10
subprogram, 8–24
terminal control, 8–16
Function (cont'd)
type of, 8–1
user-defined, 8–1, 8–17
FUNCTION statement, 11–2
format, 11–4
FUNCTION subprogram, 11–2
data type, 11–4
ending, 11–4
example of, 11–7
exiting, 11–4
naming, 11–4
parameter, 11–4

G

GET statement, 12–16 to 12–18
GETRFA function, 12–25
GOSUB statement, 6–17, 11–2

H

Handler
See Error handler
Help
EDT editor, 2–1
environment, 1–17
prompt, 1–17
topics, 1–17
HELP command, 1–17
Hyphen (-)
EXTRACT command, 1–16

I

I- and D-Space, 16–1 to 16–4
I/O
buffer, 12–6
device-specific, 17–18
matrix, 10–17
network, 17–41
RMS, 17–2
terminal-format file, 5–14
to disks, 17–34
to magnetic tape, 17–2, 17–19
to remote nodes, 17–41
to unit record devices, 17–19

%IDENT directive, 14–4
IDENTIFY command, 1–17
IF modifier, 6–1
IF statement
  in immediate mode statements, 1–9
%IF-%THEN-%ELSE-%END %IF directive, 14–8, 14–10
IF...THEN...ELSE statement, 6–10 to 6–11
Immediate mode, 1–8 to 1–9
  examining variables in, 1–8
  FOR statement in, 1–9
  IF statement in, 1–9
  invalid statements, 1–9
  UNLESS statement in, 1–9
  UNTIL statement in, 1–9
  WHILE statement in, 1–9
%INCLUDE directive, 7–16, 14–7
Indexed keys, 12–4
Initialization file
  creating, 1–26
  definition of, 1–26
Input, 5–1 to 5–8
  from source program, 5–5 to 5–8
  from terminal, 5–4
  from terminal-format files, 5–4
  interactive, 5–1
  receiving, 5–1
  string, 5–4
INPUT LINE statement, 5–4, 5–14
  prompt, 5–4
  with strings, 9–3
INPUT statement, 5–1 to 5–3, 5–14
  prompt, 5–4
  with strings, 9–3
INQUIRE command, 1–17
Instruction and data space
  See I- and D-Space
INT function, 8–2
Integer
  data type, 7–3
  format, 13–3
  output, 5–12
  variables, 4–10
INTEGER data type, 4–6
INV function, 10–22
ITERATE statement, 6–15 to 6–16

K

Keypad mode editing, 2–1
KILL statement, 12–28

L

Label, 4–2
LEN function, 9–9
%LET directive, 14–8, 14–9
LET statement, 10–7, 10–18
  dynamic strings, 9–2
  string data, 9–6
Lexical constant, 14–9
  assigning values, 14–8
  creating, 14–8
Lexical expression, 14–8, 14–9
Lexical function
  built-in, 14–9
Librarian Utility, 18–4
Library, 18–1 to 18–13
  APR, 18–11
  clustering, 18–11
  creating, 18–4
  default, 18–1
  defining, 18–5
  definition of, 18–1
  DSKLIB command, 18–5
  identifying need for, 18–3
  Librarian Utility, 18–4
  memory-resident, 1–13, 1–17, 18–1
  object module, 1–15, 18–1, 18–3
  ODL file, 18–5
  OTS routines, 18–3
  remote file access, 18–12
  RMS, 18–7
  RMS memory-resident, 18–8
  RMS object module, 18–8
  RMS ODL files, 18–9
  selecting, 18–5, 18–6
  thread names, 2–2
  types of, 18–1

Library (cont'd)
user-created, 18–1, 18–3
using, 18–5
LIBRARY command, 1–17
Line
continuing, 1–3
deleting, 1–15
displaying, 1–18
long, 1–3
Line mode, 1–8
line numbers in, 1–8
Line mode editing, 2–1
Line number
environment, 1–2
error, 15–5
error handling, 15–11
generating, 1–22
in programs, 4–1
resequencing, 19–13
rules of use, 4–1
SEQUENCE command, 1–22
using, 1–2
Line numbers
included file, 14–7
Line terminator
accepting as input, 5–4
LINK command
/BASIC qualifier, 2–13
CMD file, 2–13
file specification, 2–13
format, 2–13
ODL file, 2–13
specifying, 2–13
LINPUT statement, 5–4, 5–14
prompt, 5–4
with strings, 9–3
LIST command, 1–18
%LIST directive, 14–5
Listing file
altering, 14–2
compiler directives, 14–2
controlling, 14–5
cross-reference, 14–6
Optimizer Utility, 19–5
paging, 14–4
Listing file (cont'd)
subtitle, 14–3
title, 14–2
version number, 14–4
LISTNH command, 1–18
LOAD command, 1–6, 1–18
LOCK command, 1–19
LOG10 function, 8–4
Logarithm, 8–4
Longword
format, C–10
LONGWORD data type, 4–6
Loop, 6–3 to 6–9
control variable, 6–4
index, 6–4
Loop block, 6–3
Loops
FOR...NEXT, 6–4 to 6–7
UNTIL...NEXT, 6–8
WHILE...NEXT, 6–7
LSET statement
concatenating strings, 9–2
dynamic strings, 9–2
string data, 9–7

M

Macro subprogram, 11–25, 11–30 to 11–45
building, 11–42
calling, 11–30
common area, 11–38
debugging, 3–1
error handling, 11–45, 15–4
example of, 11–37
map area, 11–38
parameter, 11–30
virtual array, 11–31
Magnetic tape files
opening, 17–22
Map, 11–12
area, 11–13
data type, 11–14
defining area, 7–15
initializing, 11–41
multiple, 7–12, 9–18
name, 11–14

Map (cont'd)
numeric variable, 11–16
PSECT, 11–16
record buffer, 7–13
single, 7–10
storage, 11–15
string data, 9–18
MAP DYNAMIC statement, 7–17, 12–7
MAP statement, 7–10, 10–7
example of, 11–14
format, 11–14
overlaying array, 10–7
with subprograms, 7–14
MAT INPUT statement, 10–13, 10–14
prompt, 10–13
subscripts, 10–13
MAT LINPUT statement, 10–15
MAT PRINT statement, 10–16
comma (,), 10–16
semicolon (;), 10–16
MAT READ statement, 10–12
MAT statement, 10–5, 10–10
array dimensions, 10–9
keywords, 10–10
subscripts, 10–11
subtracting elements of arrays, 10–20
Matrix, 10–1
addition, 10–20
arithmetic, 10–19
assignment, 10–20
determinant return, 10–23
functions, 10–19, 10–21 to 10–23
I/O functions, 10–17
inversion, 10–19, 10–22
multiplication, 10–20
operators, 10–19
subtraction, 10–20
transposition, 10–19, 10–22
MCR commands, xxi
Memory
clearing, 1–22
extending, 20–9
large tasks, 19–1
overlaying, 11–20
saving, 19–4
Memory-resident library, 18–1
clustering, 18–11
creating, 18–4
default, 18–2
RMS, 18–7
RMS ODL file, 18–9
selecting, 18–2, 18–4
Message
compilation, 15–2
compile-time error, A–1 to A–53
error text, 15–6
run-time error, B–1 to B–32
MID$ function, 9–14
Mixed-mode expressions, 7–7
Mode
immediate, 1–8
line, 1–8
Modifiers
statement, 6–1 to 6–3
Module names, 4–3
MOVE statement, 12–6, 12–9 to 12–11
FILL formats, 7–13
Multiplication
array, 10–20

N

NAME...AS statement, 12–27
Native mode file, 12–1, 12–3
Negative field, 13–10
Network I/O, 17–41
NEW command, 1–19
%NOCROSS directive, 14–6
NOECHO function, 8–17
Nokeypad mode editing, 2–1
%NOLIST directive, 14–5
Nonprinting characters, 4–4
Notation
credit, 13–12
debit, 13–12
Null
character, 9–3
string, 9–3
NUM function, 10–17

NUM$ function, 8–8
NUM1$ function, 8–9
Number
  decimal point, 13–5
  digits, 13–4
  printing, 13–4
  special symbols in, 13–6
Numeric data
  interpreting with multiple maps, 7–13

O

Object module, 1–10
  loading, 1–18
  reducing size of, 19–1
Object module file, 2–13
Object module library, 18–3
  advantages, 18–3
  creating, 18–5
  module names in, 4–3
  RMS, 18–9
  RMS ODL file, 18–9
  selecting, 18–3, 18–5
  types of, 18–3
Object Time System
  See OTS routines
ODL file, 2–2
  default, 1–19
  editing, 11–23
  generating, 2–12
  modifying, 18–6
  overlay structure, 11–20
  RMS, 18–9, 18–10
  subprogram, 11–20
  user-created library, 18–6
ODLRMS command, 1–19
OLD command, 1–4, 1–20
ON ERROR GO BACK statement, 15–6, 15–9, 15–11
ON ERROR GOTO 0 statement, 15–10
ON ERROR GOTO statement, 15–3
ON...GOSUB...OTHERWISE statement, 6–18
ON...GOTO...OTHERWISE statement, 6–10
Online documentation, 1–17
OPEN statement, 5–14, 12–12 to 12–13
  ACCESS clause, 12–23
  ALLOW clause, 12–23
  BUCKETSIZE clause, 12–37
  BUFFER clause, 12–39
  clauses, 12–12, 12–37 to 12–46
  CLUSTERSIZE clause, 12–39
  CONNECT clause, 12–39
  CONTIGUOUS clause, 12–40
  DEFAULTNAME clause, 12–40
  EXTENDSIZE clause, 12–41
  FILESIZE clause, 12–41
  FOR INPUT, 12–12
  FOR OUTPUT, 12–12
  MAP clause, 12–12
  NOSPAN clause, 12–42
  opening indexed files, 12–13
  ORGANIZATION clause, 12–5, 12–12, 12–28
  RECORDSIZE clause, 12–12
  RECORDTYPE clause, 12–28, 12–42
  specifying file characteristics, 12–12
  TEMPORARY clause, 12–43
  UNLOCK EXPLICIT clause, 12–24
  USEROPEN clause, 12–43
  WINDOWSIZE clause, 12–46
Operand, 4–12
  promotion, 7–7
OPT command, 19–1
  /[NO]LIST qualifier, 19–2
  /[NO]OUTPUT qualifier, 19–2
  /SEGMENT_SIZE qualifier, 19–2
Optimizer Utility
  dialogue, 19–3
  error messages, 19–9
  listing file, 19–5
OPTION statement, 1–14, 7–4
OTS routines
  in library, 18–3
  object module library, 18–5
  Optimizer Utility, 19–1
  thread name, 2–2
Output, 5–8 to 5–14
  displaying, 5–8

Output (cont'd)
floating-point numbers, 5–13
format for numbers, 5–12
formatting, 5–9, 13–1
left-justified, 13–14
number format, 5–12
right-justified, 13–14
string format, 5–12
Overlay description language file
See ODL file
Overlay structure
branches, 11–20
common area, 11–17
defining, 11–20
map area, 11–17
memory requirements, 11–22
ODL file, 11–20
root, 11–20

P

%PAGE directive, 14–4
Parameter
actual, 11–8
formal, 11–8
list, 8–18, 11–8
modifiable, 11–8
passing, 11–8
passing mechanisms, 11–25
types of, 11–8
unmodifiable, 11–9
virtual array, 11–12
PLACE$ function, 8–11, 8–13
POS function, 9–10
Precision
string arithmetic, 8–11
Predefined constants
BEL, 4–8
BS, 4–8
CR, 4–8
DEL, 4–8
ESC, 4–8
FF, 4–8
HT, 4–8
LF, 4–8
Predefined constants (cont'd)
PI, 4–8
SI, 4–8
SO, 4–8
SP, 4–8
VT, 4–8
%PRINT directive, 14–10
PRINT statement, 5–8, 5–14
for array elements, 10–19
precision, 5–13
PRINT USING notation
credit, 13–12
PRINT USING statement, 13–1 to 13–18
asterisk (*), 13–8
blank-if-zero, 13–12
centered output, 13–15
comma in, 13–7
currency symbol, 13–9
debit, 13–12
E format, 13–10
error conditions, 13–17
extended field, 13–15
leading zeros, 13–11
left-justified output, 13–14
negative field, 13–10
quotation mark, 13–15
right-justified output, 13–14
string, 13–15
strings, 13–12
Print zones, 5–9 to 5–12
PROD$ function, 8–11, 8–14
Program
See also Subprogram
aborting, 14–9
appending, 1–13
array sharing, 10–6
branch, 11–20
comments, 4–4
compiling, 1–14, 2–2
conditionals, 14–10, 15–3
continuing lines, 4–2
control, 6–1 to 6–21, 15–3
controlling execution of, 14–8
creating, 1–2, 1–3 to 1–4, 2–1
creating variables, 4–5

Program (cont'd)
  Ctrl/C, 15–7
  debugging, 3–1
  definition of, 4–1
  developing, 2–1 to 2–14
  displaying, 1–18
  documenting, 4–4
  editing, 2–1
  elements of, 4–1
  error handling in, 15–2
  executing, 1–4, 2–14
  execution, 6–19
  identifying errors in, 15–4
  including file in, 14–7
  input, 5–1
  inserting message in, 14–10
  labels in, 4–2
  line, 1–3
  line numbers in, 4–1
  linking, 2–12 to 2–14
  listing, 1–18, 14–2
  main, 11–2, 11–7
  memory-resident library, 18–2
  module, 11–1
  multi-unit, 11–1
  naming, 1–19, 4–3
  no line numbers in, 4–1
  non-BASIC, 11–25
  optimization, 20–1 to 20–9
  optimizing, 19–1
  output, 5–1
  overlay structure, 11–21
  overlaying, 11–20
  PSECT, 11–13
  renaming, 1–20
  replacing, 1–20
  resequencing lines, 19–13
  root, 11–20
  running, 1–20, 2–14
  section, 11–13
  segment size, 19–2
  segmentation, 11–1
  sharing code, 14–8
  specifying data types, 4–5
  statements in, 4–3

Program (cont'd)
  subtitle, 14–3
  threaded code, 19–5
  title, 14–2
  transporting, 20–1
  units, 4–3
  version number, 14–4
PROGRAM statement, 4–3
Promotion, 7–7
Prompt
  question mark, 5–4
PSECT, 2–2, 11–13, 11–38, 19–1
  common, 11–16
  map, 11–16
  size, 11–16
PUT statement, 12–18 to 12–20
  sequential, 12–18

Q

QUO$ function, 8–11
Quotation mark
  double, 5–12
  PRINT statement, 5–12
  single, 5–12

R

Radix–50
  See RAD-50
Random access, 12–4
Random number generator, 8–5
RANDOMIZE statement, 8–6
RCTRLC function, 8–16
READ statement, 5–6 to 5–7
REAL data type, 4–6
Record
  access, 12–5
  buffer, 7–13, 12–6
  by key access, 12–5
  by RFA access, 12–5
  channel number, 12–37
  character transfer, 12–36
  context, 12–5
  controlling access, 12–23
  current, 12–5

Record (cont'd)
deleting, 12–20
file address, 12–24
fixed-length, 12–2
format, 12–2 to 12–3
locating, 12–15
locked, 12–18
locking, 12–24
MOVE statement, 12–9
moving variables, 12–9
next, 12–5
operations, 12–11
order, 12–5
pointer, 12–5
after file location, 12–15
after file retrieval, 12–16
after update, 12–22
resetting, 12–26
random access, 12–5, 12–16
reading, 12–16
retrieving, 12–16
RFA access, 12–24
sequential access, 12–5
stream format, 12–3
unlocking, 12–24
updating, 12–21, 12–24
variable-length, 12–2
variables, 12–9
writing, 12–18
Record File Address (RFA), 12–24
Record Management Services
See RMS
RECOUNT function, 12–36
Relative file, 12–4
REM statement, 4–5
REMAP statement, 7–17, 12–7
FILL formats, 7–13
Remote files
accessing, 17–41
RENAME command, 1–20
REPLACE command, 1–7, 1–20
Resequencer Utility, 19–13
command file, 19–15
commands, 19–15
error messages, 19–16
RESTORE # statement, 11–17
RESTORE statement, 5–7, 12–26
RESUME statement, 15–6, 15–11
Retrieval pointers, 12–46
RETURN statement, 6–17
RFA data type, 4–6
RMS, 17–1
libraries, 18–7
ODL file, 18–9
using, 12–1
RMSRES command, 18–8
RND function, 8–5
Root, 11–20
Round-off errors
overcoming with SCALE command, 1–22
Row, 10–16
RSET statement
concatenating strings, 9–2
dynamic strings, 9–2
string data, 9–7
RUN command, 1–4, 1–10, 1–20, 2–14
Run-time error, 15–1
cause of, 2–14
list of, B–1 to B–32
RUNNH command, 1–4, 1–20

S

SAVE command, 1–21
%SBTTL directive, 14–3
SCALE command, 1–22
SCRATCH command, 1–22
SCRATCH statement, 12–26
SEG$ function, 8–19, 9–12
Segment size, 19–2
SELECT block, 6–13
SELECT...CASE statement, 6–13 to 6–15
Semicolon (;)
MAT PRINT statement, 10–16
PRINT statement, 5–10
SEQUENCE command, 1–22
Sequential file, 12–3
SET command, 1–10, 1–23
/CLUSTER qualifier, 18–11

SET NO PROMPT statement
  prompt, 5–5
SET VARIANT command, 14–9
SET [NO] PROMPT statement, 5–4 to 5–5, 10–13
SHOW command, 1–10, 1–23
SIN function, 8–3
Single-precision
  format, 5–13, C–12
SLEEP statement, 6–19
SPACE$ function, 9–15
Statement
  declarative, 4–7
Statement modifiers, 6–1 to 6–3
  FOR, 6–1
  IF, 6–1
  UNLESS, 6–1
  UNTIL, 6–1
  WHILE, 6–1
Static storage, 4–7
  allocating, 7–9
  dynamic mapping, 7–17
STATUS function, 12–37
STEP clause, 6–4
STOP statement, 1–8, 6–19, 6–20
Storage
  dynamic, 4–7, 20–2
  redefining, 7–16
  static, 4–7, 7–17, 20–2
Stream record, 12–3
String
  arithmetic functions, 8–10
  assigning data, 9–5
  centered, 13–15
  constant, 9–1
  data formatting, 13–1
  definition of, 9–1
  dynamic, 9–1
  fixed-length, 9–1, 9–4
  format, 13–1, C–14
  format field, 13–13, 13–15
  functions, 9–9 to 9–17
  handling, 9–1 to 9–20
  left-justified, 13–14
  literal, 5–9
String (cont'd)
  manipulating, 9–9, 9–18
  manipulating with multiple maps, 7–12
  mapping storage, 9–18
  numeric, 8–8
  output, 5–12
  printing, 13–12
  right-justified, 13–14
  storage, 9–18
  variable, 4–10, 9–1
  virtual array, 9–1
  virtual arrays, 9–5
STRING data type, 4–6
STRING$ function, 9–15
SUB statement, 11–2
  format, 11–3
SUB subprogram, 11–2
  example of, 11–7
Subprogram
  array, 11–12
  array sharing, 10–6
  calling, 11–2, 11–6
  common, 11–12
  controlling execution, 14–8
  creating, 11–1
  Ctrl/C, 15–7
  data type, 11–5
  debugging, 3–1
  declaring, 11–5
  definition of, 11–1
  ending, 11–4
  environment, 1–6
  error handling, 15–4, 15–6, 15–8
  error handling in, 15–2
  executing, 11–17
  exiting, 11–4
  file access, 11–17
  function, 8–24
  FUNCTION statement, 11–2
  invoking, 11–6, 11–23
  linking, 11–18
  macro, 11–25
  map, 11–12
  name, 11–3, 11–6
  non-BASIC, 11–24 to 11–45

Subprogram (cont'd)
overlay structure, 11–21
overlaying, 11–17
parameter, 11–3, 11–9
parameter passing, 11–8
sharing code, 14–8
sharing data, 5–7, 11–12
SUB statement, 11–2
types of, 11–2
Subroutine, 11–2
definition of, 6–16
entry point, 6–17
executing, 6–16
local, 6–16
Subscripted variables, 4–9
Subscripts, 4–10
in MAT READ statement, 10–12
Subtraction
array, 10–20
SUM$ function, 8–11

T

TAN function, 8–3
Tape unit
allocating for device-specific I/O, 17–20
Tapes
allocating, 17–2
Task
executing large, 19–1
extending, 16–1
I- and D-Space, 16–1
image file, 2–12
memory requirements, 11–22
overlaying, 11–20
Task Builder
CMD file, 1–13, 2–2
function, 2–2
invoking, 2–12, 2–13
LINK command, 2–13
ODL file, 1–13, 2–2
RMS library, 18–8
RUN $TKB, 2–12
subprogram, 11–18
task image file, 2–12
task overlay, 11–23
Task Builder (cont'd)
TKB command, 2–12
Terminal control
functions, 8–16
Terminal-format file, 12–3, 12–26
closing, 5–14
I/O, 5–14
opening, 5–14
writing records to, 5–14
THEN clause, 6–10
Thread, 2–2, 19–1
TIME function, 8–15
TIME$ function, 8–15
%TITLE directive, 14–2
TKB command, 2–12
CMD file, 2–12
ODL file, 2–12
TRM$ function, 9–16
TRN function, 10–22

U

UNLESS modifier, 6–1
UNLESS statement
in immediate mode statements, 1–9
UNLOCK statement, 12–24
UNSAVE command, 1–25
UNTIL modifier, 6–1
UNTIL statement
in immediate mode statements, 1–9
UNTIL...NEXT loops, 6–8
UPDATE statement, 12–21 to 12–23
User-created library, 18–3
Utility
Resequencer, 19–13

V

VAL function, 8–10
VAL% function, 8–10
Variable
array, 4–9, 4–10, 12–9
changing value of, 14–8
control, 6–4
data type, 7–5
declaring, 4–5

Variable (cont'd)
FILL, 12–9
floating-point, 4–9
initialization, 1–9, 4–11, 7–10, 15–12, 20–3
integer, 4–10
loop, 6–4
names, 4–9
record, 12–9
redefining, 7–16
scalar, 12–9
string, 4–10, 9–1
subscripted, 4–9, 4–10
Variable-length record, 12–2
%VARIANT directive, 14–8, 14–9
Vector, 10–1
Virtual array
macro subprogram, 11–31
string, 9–1, 9–5
Virtual array file, 12–5
creating, 12–14

W

WAIT statement, 6–19, 6–20
Warning error, 15–2, A–1
WHILE modifier, 6–1
WHILE statement
in immediate mode statements, 1–9
WHILE...NEXT loops, 6–7
Word
array descriptor, C–19
boundary, 11–16
format, C–10
WORD data type, 4–6

Z

Zero
division by, 15–3

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

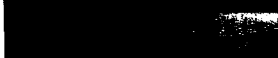
Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal[1]	————	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).



Reader's Comments

BASIC–PLUS–2
User's Guide

AA–JP35B–TK

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ______________________________

____

What I like best about this manual is ______________________________

____

What I like least about this manual is ______________________________

____

I found the following errors in this manual:

Page Description

______ ______________________________

______ ______________________________

______ ______________________________

Additional comments or suggestions to improve this manual:

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ______________________ Dept. ______________

Company ______________________ Date ______________

Mailing Address ______________________________

______________________ Phone ______________

— 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

DIGITAL EQUIPMENT CORPORATION
Corporate User Information Products
NUO1-1/G10
55 NORTHEASTERN BLVD
NASHUA, NH 03062-9934

-- Do Not Tear - Fold Here