BLISS–32

User Manual

Order Number: AA–H322E–TE

May 1987

This document describes the BLISS–32 compiler and provides basic information about linking, executing, and debugging BLISS–32 programs. It also describes BLISS–32 machine-specific functions, BLISS tools, and other topics related to BLISS–32 programming.

Revision/Update Information: This book supersedes the BLISS–32 User's Guide (Order No. AA–H322D–TE).

Operating System and Version: VMS Version 4.2 or higher

Software Version: BLISS–32 Version 4.3

Digital Equipment Corporation
Maynard, Massachusetts

First Printing, January 1980
Revised, February 1982
Revised, November 1983
Revised, May 1987

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Digital or an authorized sublicensor.

© Digital Equipment Corporation 1980, 1982, 1983, 1987. All rights reserved.

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: DEC, DEC/CMS, DEC/MMS, DECnet, DECsystem–10, DECSYSTEM–20, DECUS, DECwriter, DIBOL, EduSystem, IAS, MASSBUS, PDP, PDT, RSTS, RSX, UNIBUS, VAX, VAXcluster, VMS, VT, and the DIGITAL logo.

ZK4359

This document is available on CD–ROM.

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

Preface xi

Summary of Technical Changes xiii

1 Operating Procedures

1.1 Compiling a BLISS Module 1–1
1.1.1 Command-Line Syntax 1–2
1.1.2 Command-Line Semantics 1–2
1.2 File Specifications 1–3
1.3 Command-Line Qualifiers 1–4
1.3.1 /CHECK Qualifier 1–5
1.3.1.1 Syntax 1–5
1.3.1.2 Defaults 1–5
1.3.1.3 Semantics 1–5
1.3.2 /CROSS_REFERENCE Qualifier 1–6
1.3.2.1 Syntax 1–7
1.3.2.2 Defaults 1–7
1.3.2.3 Semantics 1–7
1.3.3 General Qualifiers 1–7
1.3.3.1 Syntax 1–8
1.3.3.2 Defaults 1–8
1.3.3.3 Semantics 1–8
1.3.3.4 Discussion 1–9
1.3.4 /MACHINE_CODE_LIST Qualifier 1–9
1.3.4.1 Syntax 1–9
1.3.4.2 Defaults 1–10
1.3.4.3 Semantics 1–10
1.3.5 /OPTIMIZE Qualifier 1–11
1.3.5.1 Syntax 1–11
1.3.5.2 Defaults 1–12
1.3.5.3 Semantics 1–12
1.3.6 Output Qualifiers 1–13
1.3.6.1 Syntax 1–13
1.3.6.2 Defaults 1–14
1.3.6.3 Semantics 1–14
1.3.7 /SOURCE_LIST Qualifier 1–14
1.3.7.1 Syntax 1–15
1.3.7.2 Defaults 1–15
1.3.7.3 Semantics 1–15
1.3.8 /TERMINAL Qualifier 1–16
1.3.8.1 Syntax 1–16
1.3.8.2 Defaults 1–16
1.3.8.3 Semantics 1–17

1.3.9 Qualifier Names Versus Switch Names 1–17
1.3.10 Qualifiers and Default Settings 1–17
1.3.11 Positive and Negative Forms of Qualifiers 1–18
1.3.12 Abbreviations of Qualifier and Value Names 1–18

2 Compiler Output

2.1 Terminal Output 2–2
2.2 Output Listing 2–3
2.2.1 Listing Header 2–4
2.2.2 Source Listing 2–5
2.2.3 Object Listing 2–7
2.2.3.1 Default Object Listing 2–8
2.2.3.2 Assembler Input Listing 2–9
2.2.4 Source Part Options 2–13
2.2.4.1 Default Source Listing 2–14
2.2.4.2 Listing with LIBRARY and REQUIRE Information 2–14
2.2.4.3 Listing with Macro Expansions 2–14
2.2.4.4 Listing with Macro Tracing 2–14
2.3 Cross-Reference Listing 2–19
2.3.1 Cross-Reference Header 2–19
2.3.2 Cross-Reference Entries 2–19
2.3.3 Output Listing with Cross-Reference Listing 2–24
2.4 Compilation Summary 2–27
2.5 Error Messages 2–27

3 Linking, Executing, and Debugging

3.1 Linking 3–1
3.1.1 LINK Command Syntax 3–1
3.1.2 LINK Command Semantics 3–1
3.2 Executing a Linked Program 3–2
3.3 Debugging 3–2
3.3.1 Initialized Modes and Types 3–3
3.3.2 Debugger Commands and Expression Syntax 3–3
3.3.3 Operators in Arithmetic Expressions 3–4
3.3.4 Special Characters in Address Expressions 3–5
3.3.4.1 Current Location Symbol (.) 3–5
3.3.4.2 Last Value Displayed Symbol (\) 3–6
3.3.4.3 Contents Operator (.) 3–6
3.3.4.4 Range Operator (:) 3–6
3.3.4.5 Default Next Location Value 3–7
3.3.5 Field References 3–8
3.3.6 Structure References 3–9
3.3.7 REF Structure References 3–11
3.3.8 Scope of Names 3–12
3.3.9 Source-Line Debugging 3–12
3.3.10 Effect of Compilation and Link-Time Qualifiers 3–13
3.3.11 Debugger Command Summary 3–13

4 Machine-Specific Functions

4.1 ADAWI—Add Aligned Word Interlocked 4–6
4.2 ADDD—Add D-Floating Operands 4–7
4.3 ADDF—Add F-Floating Operands 4–7
4.4 ADDG—Add G-Floating Operands 4–7
4.5 ADDH—Add H-Floating Operands 4–8
4.6 ADDM—Add Multiword Operands 4–8
4.7 ASHP—Arithmetic Shift and Round Packed 4–8
4.8 ASHQ—Arithmetic Shift Quad 4–9
4.9 BICPSW—Bit Clear PSW 4–10
4.10 BISPSW—Bit Set PSW 4–10
4.11 BPT—Break Point Trap 4–10
4.12 BUGL—Bugcheck with Long Operand 4–10
4.13 BUGW—Bugcheck with Word Operand 4–11
4.14 CALLG—Call with General Parameter List 4–11
4.15 CHMx—Change Mode 4–12
4.16 CMPC3—Compare Characters 3 Operand 4–12
4.17 CMPC5—Compare Characters 5 Operand 4–13
4.18 CMPD—Compare D-Floating Operands 4–14
4.19 CMPF—Compare F-Floating Operands 4–14
4.20 CMPG—Compare G-Floating Operands 4–14
4.21 CMPH—Compare H-Floating Operands 4–15
4.22 CMPM—Compare Multiword Operands 4–15
4.23 CMPP—Compare Packed 4–16
4.24 CRC—Cyclic Redundancy Check 4–16
4.25 CVTDF—Convert D-Floating to F-Floating 4–17
4.26 CVTDI—Convert D-Floating to Integer 4–17
4.27 CVTDL—Convert D-Floating to Long 4–17
4.28 CVTFD—Convert F-Floating to D-Floating 4–18
4.29 CVTFG—Convert F-Floating to G-Floating 4–18
4.30 CVTFH—Convert F-Floating to H-Floating 4–18
4.31 CVTFI—Convert F-Floating to Integer 4–19
4.32 CVTFL—Convert F-Floating to Long 4–19
4.33 CVTGF—Convert G-Floating to F-Floating 4–19
4.34 CVTGH—Convert G-Floating to H-Floating 4–20
4.35 CVTGL—Convert G-Floating to Long 4–20
4.36 CVTHF—Convert H-Floating to F-Floating 4–20
4.37 CVTHG—Convert H-Floating to G-Floating 4–20
4.38 CVTHL—Convert H-Floating to Long 4–21
4.39 CVTID—Convert Integer to D-Floating 4–21
4.40 CVTIF—Convert Integer to F-Floating 4–21
4.41 CVTLD—Convert Long to D-Floating 4–22
4.42 CVTLF—Convert Long to F-Floating 4–22
4.43 CVTLH—Convert Long to H-Floating 4–22
4.44 CVTLP—Convert Long to Packed 4–23
4.45 CVTPL—Convert Packed to Long 4–23
4.46 CVTPS—Convert Packed to Leading Separate Numeric 4–23
4.47 CVTPT—Convert Packed to Trailing Numeric 4–24
4.48 CVTRDL—Convert Rounded D-Floating to Long 4–25
4.49 CVTRFL—Convert Rounded F-Floating to Long 4–25
4.50 CVTRGL—Convert Rounded G-Floating to Long 4–25
4.51 CVTRHL—Convert Rounded H-Floating to Long 4–26
4.52 CVTSP—Convert Leading Separate to Packed 4–26
4.53 CVTTP—Convert Trailing Numeric to Packed 4–27

4.54 DIVD—Divide D-Floating Operands 4–27
4.55 DIVF—Divide F-Floating Operands 4–28
4.56 DIVG—Divide G-Floating Operands 4–28
4.57 DIVH—Divide H-Floating Operands 4–28
4.58 EDITPC—Edit Packed to Character 4–29
4.59 EDIV—Extended-Precision Divide 4–30
4.60 EMUL—Extended-Precision Multiply 4–30
4.61 FFC and FFS—Find and Modify Operations 4–30
4.62 HALT—Halt Processor 4–31
4.63 INDEX—Index Calculation 4–31
4.64 INSQHI and INSQTI—Insert Entry in Queue, Interlocked 4–32
4.65 INSQUE—Insert Entry in Queue 4–32
4.66 LOCC—Locate Character 4–33
4.67 MATCHC—Match Characters 4–33
4.68 MFPR—Move from Processor Register 4–34
4.69 MOVC3—Move Character 3 Operand 4–34
4.70 MOVC5—Move Character 5 Operand 4–35
4.71 MOVP—Move Packed 4–36
4.72 MOVPSL—Move from PSL 4–36
4.73 MOVTC—Move Translated Characters 4–37
4.74 MOVTUC—Move Translated Until Character 4–37
4.75 MTPR—Move to Processor Register 4–38
4.76 MULD—Multiply D-Floating Operands 4–39
4.77 MULF—Multiply F-Floating Operands 4–39
4.78 MULG—Multiply G-Floating Operands 4–39
4.79 MULH—Multiply H-Floating Operands 4–40
4.80 NOP—No Operation 4–40
4.81 PROBER—Probe Read Accessibility 4–40
4.82 PROBEW—Probe Write Accessibility 4–41
4.83 REMQHI and REMQTI—Remove Entry From Queue, Interlocked 4–41
4.84 REMQUE—Remove Entry from Queue 4–42
4.85 ROT—Rotate a Value 4–42
4.86 SCANC—Scan Characters 4–43
4.87 SKPC—Skip Character 4–43
4.88 SPANC—Span Characters 4–44
4.89 SUBD—Subtract D-Floating Operands 4–44
4.90 SUBF—Subtract F-Floating Operands 4–45
4.91 SUBG—Subtract G-Floating Operands 4–45
4.92 SUBH—Subtract H-Floating Operands 4–45
4.93 SUBM—Subtract Multiword Operands 4–46
4.94 TESTBITx—Test and Modify Operations 4–46
4.95 XFC—Extended Function Call 4–47

5 Programming Considerations

5.1 Library and Require File Differences 5–1
5.2 Frequent BLISS Coding Errors 5–2
5.2.1 Missing Dots 5–3
5.2.2 Valued and Nonvalued Routines 5–3
5.2.3 Semicolons and Values of Blocks 5–3
5.2.4 Complex Expressions Using AND, OR, and NOT 5–4
5.2.5 Computed Routine Calls 5–4
5.2.6 Signed and Unsigned Fields 5–4
5.2.7 Complex Macros 5–5

5.2.8 Missing Code 5–5
5.2.9 Using the Built-In PC 5–5
5.2.10 Indexed Loop Coding Error 5–6
5.3 Linker Error Messages 5–7
5.4 Obscure Error Messages 5–7
5.5 Position-Independent Code Generation 5–7
5.6 Advanced Use of BLISS Macros 5–7
5.6.1 Advantageous Use of Machine Dependencies 5–7
5.6.2 Using String Instructions 5–8
5.6.3 Dealing with Enumeration Types 5–9
5.6.3.1 The SET Data Type 5–9
5.6.3.2 Creating a Set 5–9
5.6.3.3 Placing Elements in Sets 5–11
5.6.3.4 Membership in a Set 5–12

6 Transportability Guidelines

6.1 Introduction 6–1
6.2 General Strategies 6–2
6.2.1 Isolation 6–2
6.2.2 Simplicity 6–3
6.3 Tools 6–3
6.3.1 Literals 6–3
6.3.1.1 Predeclared Literals 6–3
6.3.1.2 User-Defined Literals 6–4
6.3.2 Macros and Conditional Compilation 6–5
6.3.3 Module Switches 6–5
6.3.4 Reserved Names 6–7
6.3.5 REQUIRE and Library Files 6–8
6.3.6 Routines 6–9
6.4 Techniques for Writing Transportable Programs 6–10
6.4.1 Allocation of Data 6–10
6.4.1.1 Problem Origin 6–10
6.4.1.2 Transportable Declarations 6–11
6.4.2 Data Addresses 6–12
6.4.2.1 Addresses and Address Calculations 6–12
6.4.2.2 Relational Operators and Control Expressions 6–14
6.4.2.3 BLISS–10 Addresses Versus BLISS–36 Addresses 6–14
6.4.3 Data Character Sequences 6–15
6.4.3.1 Quoted Strings Used as Numeric Values 6–15
6.4.3.2 Quoted Strings Used as Character Strings 6–16
6.4.4 PLITs and Initialization 6–16
6.4.4.1 PLITs in General 6–17
6.4.4.2 Scalar PLIT Items 6–17
6.4.4.3 String Literal PLIT Items 6–17
6.4.4.4 Initialization Example 6–19
6.4.4.5 Initializing Packed Data 6–22
6.4.5 Structures and Field Selectors 6–25
6.4.5.1 Structures 6–25
6.4.5.2 FLEX_VECTOR 6–25
6.4.5.3 Field Selectors 6–28
6.4.5.4 GEN_VECTOR 6–28
6.4.5.5 Summary of Structures and Field Selectors 6–30

7 Compiler Overview and Optimization Switches

7.1 Compiler Phases 7–1
7.1.1 Lexical and Syntactic Analysis Phase 7–1
7.1.2 Flow Analysis Phase 7–2
7.1.2.1 Knowing When a Value Changes 7–3
7.1.2.2 Accounting for Changes 7–4
7.1.3 Heuristic Phase 7–6
7.1.4 Temporary Name Binding Phase 7–6
7.1.5 Code Generation Phase 7–7
7.1.6 Code Stream Optimization Phase 7–7
7.1.7 Output File Production Phase 7–7
7.2 Summary of Switch Effects 7–7

8 Tools, Libraries, and System Interfaces

8.1 VAX Source Code Analyzer (SCA) 8–1
8.2 Tutorial Terminal Input/Output Package (TUTIO) 8–1
8.3 VMS System Services Interface 8–2
8.3.1 Sample Program Using VMS System Services 8–2
8.3.2 Common Errors in Using System Services 8–3
8.4 Record Management Services Interface 8–3
8.4.1 Using VAX RMS Macros 8–4
8.4.2 Sample Routine Using VAX RMS 8–4
8.5 Other VMS Interfaces 8–5
8.5.1 LIB Interface 8–5
8.5.2 TPAMAC Interface 8–5

9 BLISS–32 Code Examples

9.1 Example 1: The ALPHA Program 9–1
9.1.1 Module ALPHA 9–2
9.1.2 Routine SORT_CHARS 9–3
9.1.3 Routine ALPHA_SORT 9–5
9.1.4 Cell-Handling Package 9–6
9.1.5 Macro REL_CELL 9–7
9.1.6 Macro INS_CELL 9–8
9.1.7 Macro GET_CELL 9–9
9.1.8 Macro REM_CELL 9–9
9.2 Example 2: The CALC Program 9–10
9.2.1 Module CALCULATE 9–11
9.2.2 Routine CALC_DRIVER 9–12
9.2.3 Routine MAIN 9–14
9.2.4 Routine READEXPRESSION 9–15
9.2.5 Routine READTERM 9–16
9.2.6 Routine READFACTOR 9–17
9.2.7 Routine READNUMBER 9–18
9.2.8 Routine READCHAR 9–19
9.2.9 Routine CATCH_ERRORS 9–19

A Summary of Command Syntax

A.1 Command-Line Syntax A–1
A.2 File Specification Summary A–1
A.3 Qualifier Syntax A–1
A.4 Qualifier Defaults A–3
A.5 Abbreviations A–3

B Summary of Formatting Rules

C Module Template

C.1 Module Preface C–1
C.2 Declarative Part of Module C–2
C.3 Executable Part of Module C–2
C.4 Closing Format C–3

D Implementation Limits

D.1 BLISS–32 Constructs D–1
D.2 System Interfaces D–1

E Error Messages

E.1 BLISS Compiler Fatal Errors E–28

F Sample Output Listing

G Optional Programming Productivity Tools

G.1 Using LSE with BLISS–32 . . . . . G–1
G.1.1 Entering Source Code Using Tokens and Placeholders G–1
G.1.2 Compiling Source Code G–3
G.1.3 Examples G–3
G.1.4 Module Declaration G–4
G.1.4.1 Routine Declaration G–5
G.1.4.2 IF Statement G–6
G.1.4.3 Select Expression G–7
G.1.4.4 CH$COMPARE Function G–9
G.2 Using the VAX Source Code Analyzer G–10
G.2.1 Setting Up an SCA Environment G–12
G.2.1.1 Creating an SCA Library G–12
G.2.1.2 Generating Data Analysis Files G–12
G.2.1.3 Loading Data Analysis Files into a Local Library G–12
G.2.1.4 Selecting an SCA Library G–12
G.2.2 Using SCA for Cross-Referencing G–13

Index

Examples

2–1	Default Object Listing	2–9
2–2	Assembler Input Listing	2–11
2–3	Default Source Listing	2–15
2–4	Output Listing with LIBRARY and REQUIRE File Data	2–16
2–5	Output Listing with Macro Expansion Data	2–17
2–6	Output Listing with Macro Expansion and Tracing Data	2–17
2–7	Output Listing with Cross-Reference Listing Included	2–24
2–8	Example of Error Messages in Source Listing	2–30
8–1	Sample TPARSE Program	8–6
F–1	Sample Output Listing	F–1

Figures

2–1	Compiler Output Listing Sequence	2–3
2–2	Listing Header Format	2–4
G–1	Use of LSE and SCA for Multimodular Development	G–11

Tables

1–1	Correspondence Between Qualifier and Switch Names	1–17
2–1	Format of Preface String in Source Listing	2–5
2–2	Symbol Type Abbreviations	2–20
3–1	Arithmetic Expression Operators	3–4
3–2	Address Representation Characters	3–5
4–1	Machine-Specific Functions	4–2

Preface

Manual Objectives

This manual is a user's guide for the VAX BLISS–32 compiler running under the VMS operating system. It is intended as a complement to the *BLISS Language Guide*. It provides three kinds of information: basic operating instructions, supplementary programming information, and reference material.

Intended Audience

This guide is intended for users of the BLISS–32 programming language. It presupposes some familiarity with the VMS operating system, its command language, and its file system conventions.

Document Structure

Chapters 1 through 3 describe basic operating instructions.

- Chapter 1 gives procedures for compiling a BLISS–32 program and describes the command qualifiers.
- Chapter 2 considers output produced by the compilation. The format and meaning of each of the possible compiler outputs are described and illustrated.
- Chapter 3 is concerned with linking, executing, and debugging.

Chapters 4 through 9 supply supplementary programming information.

- Chapter 4 defines the VAX machine-specific functions.
- Chapter 5 discusses programming considerations, such as using LIBRARY and REQUIRE facilities.
- Chapter 6 gives guidelines for writing transportable BLISS–32 programs.
- Chapter 7 describes compiler architecture and the effects of the command qualifiers related to optimization.
- Chapter 8 describes some tools related to BLISS–32 programming.
- Chapter 9 provides coding examples in the form of two complete and annotated programs.

The appendixes contain reference material.

- Appendix A summarizes the command syntax, including the command qualifiers and their defaults and abbreviations.
- Appendix B provides formatting rules.

- Appendix C provides a module template.
- Appendix D lists current implementation limits.
- Appendix E describes the error messages produced by the compiler.
- Appendix F is a sample output listing.
- Appendix G describes how to use related development tools.

Associated Documents

The *VAX Information Directory* lists and describes all of the documents you may need to refer to in the course of building and executing a BLISS program.

Additional documentation that is either directly or indirectly relevant to BLISS programming includes the following:

BLISS language usage:	*BLISS Language Reference Manual* (AA–H275D–TK)
BLISS syntax summary:	*BLISS Pocket Guide* (AV–H289D–TK)
Transportability tool:	*XPORT Programmer's Guide* (AA–201A–TK)
Program linking and execution:	*VAX/VMS Linker Reference Manual* (AA–Z420A–TE)
	VAX/VMS DCL Dictionary (AA–Z200C–TE)
Symbolic debugging:	*VAX/VMS Debugger Reference Manual* (AA–Z411C–TE)
System services:	*VAX/VMS System Services Reference Manual* (AA–Z501C–TE)
	VAX Record Management Services Reference Manual (AA–Z503B–TE)

Conventions

Syntax notation and definitions used in BLISS–32 are explained thoroughly in Chapter 2 of the *BLISS Language Guide*. The following is a summary of syntax notations used in this manual:

{ item-1 | item-2 | item-3 }	Select exactly one of the items separated by vertical bars within the braces.
{ item-1 item-2 item-3 }	Select exactly one of the items in braces on separate but contiguous lines.
item . . .	The item directly preceding the " . . . " can be repeated zero or more times with the items separated by spaces.
item, . . .	The item directly preceding the ", . . . " can be repeated zero or more times with the items separated by commas.
item+ . . .	The item directly preceding the "+ . . . " can be repeated zero or more times with the items separated by plus signs.

In addition, the red portions of a syntax line or system-user dialogue identify information entered in by the user.

Summary of Technical Changes

This manual provides BLISS–32 user information for Version 4.3 of the BLISS–32 compiler. This list summarizes technical changes, additions, and deletions to the guide since Version 4.0.

- /ANALYSIS_DATA has been added as an output-qualifier for the BLISS–32 command line.
- The VAX Source Code Analyzer (SCA) programming tool makes the /MASTER_CROSS_REFERENCE qualifier and BCREF obsolete.
- XPORT material has been removed; this information is available from the DECUS library.

1
Operating Procedures

This chapter describes the operating procedures used to compile a BLISS module. The form of the command line is considered first. Next, the file specifications for input to a BLISS–32 compilation are described and illustrated. Finally, the command-line qualifiers relevant to a BLISS–32 compilation are given.

The procedure for compiling, linking, and executing a BLISS–32 program is uncomplicated. For example, to compile and execute a program consisting of a single source module, enter the module in a file (for example, ALPHA.B32), compile it with the BLISS–32 compiler, link it using the VMS Linker, and execute the linked image. The command sequence to do this is as follows:

```
$ BLISS ALPHA
$ LINK ALPHA
$ RUN ALPHA
```

The first command invokes the BLISS compiler to compile the module in the file ALPHA.B32 and to produce an object file ALPHA.OBJ. The second command uses the object module in the file ALPHA.OBJ to produce an executable image in the file ALPHA.EXE. The third command executes the image in the file ALPHA.EXE.

However, the more usual case involves compiling and linking several (and possibly a large number of) source modules into one executable image.

You can use command-line qualifiers to control the compiler. These qualifiers add a level of complexity to the compilation process. However, they provide a means by which you can vary the performance of the compiler, for example, in the production of output, in the formatting of listings, and in the degree of optimization to be performed.

1.1 Compiling a BLISS Module

To compile a BLISS module, issue the DCL command BLISS, followed by a BLISS command line. The command line consists of one or more source file names optionally preceded by command-line qualifiers. (Refer to Section 1.1.1.) Some BLISS command line examples follow:

- To compile a module, give the following command:

  ```
  $ BLISS MYPROG
  ```

  The BLISS compiler uses file MYPROG.B32 as input, compiles the source in that file, and produces an object file, MYPROG.OBJ.

- To produce a listing file, use the /LIST output qualifier:

  ```
  $ BLISS/LIST MYPROG
  ```

  In addition to the object file, the BLISS compiler produces the listing file MYPROG.LIS.

- To produce an object file with a name different from that of the source file, give the name in the command as follows:

  ```
  $ BLISS/OBJECT=GAMMA ALPHA
  ```

  The BLISS compiler produces the object file GAMMA.OBJ.

- To produce a BLISS library file instead of an object file, use the /LIBRARY command qualifier:

  ```
  $ BLISS/LIBRARY ALPHA
  ```

  The BLISS compiler compiles the input file ALPHA.R32 and produces the library file ALPHA.L32.

- To compile more than one module, include a list of input files separated by commas:

  ```
  $ BLISS ALPHA,BETA,GAMMA
  ```

  The compiler compiles ALPHA.B32, producing the object file ALPHA.OBJ; then BETA.B32, producing BETA.OBJ; and then GAMMA.B32, producing GAMMA.OBJ.

- To compile a module that consists of several pieces, each in a separate file, use the concatenation indicator (+):

  ```
  $ BLISS ALPHA+BETA+GAMMA
  ```

  The BLISS compiler compiles the source module formed by the concatenation of ALPHA.B32, BETA.B32, and GAMMA.B32, and produces the single object file ALPHA.OBJ.

1.1.1 Command-Line Syntax

compilation-request	$ BLISS [/qualifier . . .] input-spec, . . .
input-spec	file-spec[+ . . .] [/qualifier . . .]
qualifiers	{ check qualifier general qualifiers machine-code-list qualifier optimize qualifier output qualifiers reference qualifier source-list qualifier terminal qualifier }

The dollar sign ($) represents the DCL prompt. As indicated in the syntax rule, a space must immediately precede the first or only input-spec. Optional spaces may be used before or after any delimiter character shown in this and subsequent syntax diagrams. The applicable delimiters are the comma (,), plus sign (+), slash (/), equal sign (=), and colon (:).

1.1.2 Command-Line Semantics

The BLISS–32 compiler uses command-line qualifiers given in the BLISS–command-line to modify their default settings for each compilation. Then, each input-spec is compiled separately in the context of the initial default qualifier settings. Qualifiers and their initial default settings are described in Section 1.3.

Unless you use a qualifier to change the compiler's behavior, the output from an interactive compilation is an object file and a terminal listing, and the output from a batch compilation is an object file and a listing file.

The compiler uses the contents of a file or of a series of files joined by plus signs (+) as input to a BLISS compilation. The compiler begins processing with the first file given in an input-spec and continues until an end-of-file is reached. It continues to read input until all files specified in the input-spec have been read. Command-line qualifiers can appear in two places in a command line: before the first input-spec and after individual input-specs. Those appearing before the first input-spec have a global application to all input-specs in the command-line. For example:

```
$ BLISS/LIBRARY ALPHA,BETA+THETA+ZETA,OMEGA
```

Those appearing at the end of an input-spec apply only to the input-spec they follow. For example:

```
$ BLISS/LIBRARY ALPHA,BETA/OBJECT,IOTA
```

If no command-line qualifiers exist in a command line, default qualifier settings are assumed for all input-specs in the command line. All qualifiers have an assigned default setting or value.

The only required space in the command line separates the first input-spec from preceding global command-line qualifiers.

1.2 File Specifications

File specifications are used to name the source of program text to be compiled and the destination of output from the compilation. File specifications can occur in three contexts:

- In the input-specs of a BLISS–command-line
- As the values of the qualifiers /OBJECT, /LIBRARY, /LIST, and /ANALYSIS-DATA
- In REQUIRE and LIBRARY declarations in the module being compiled

The file-spec is a standard VMS file specification, as described in the *VAX/VMS DCL Dictionary.* (See Appendix A.)

A file specification is interpreted as follows:

1. Logical names are translated.
2. If a file type is not given, a default file type is used, as described in the next section.
3. If the file-spec applies to an output file and a file name is not given, then the name of the first or only input file in the input-spec is used.

The compiler uses this same interpretation when it processes the file specification given in a REQUIRE or LIBRARY declaration. (Refer to the *BLISS Language Reference Manual*). The compiler has two ordered lists of default file types to be tried for an input-spec that does not include a file type. The default type that the compiler applies depends on the output to be produced by the compilation, as indicated in the following list:

Input-spec Used to Produce	Default Type List
An object module	.B32, .BLI
A library file	.R32, .REQ, .B32, .BLI

If the program being compiled contains a REQUIRE or LIBRARY declaration, the compiler uses the following lists to search for the appropriate file type according to the type of declaration:

File Use	Default Type List
File given in a REQUIRE declaration	.R32, .REQ, .B32, .BLI
File given in a LIBRARY declaration	.L32

The default file-types for output files that the compiler generates are as follows:

File-Designator	Default Type
/OBJECT	.OBJ
/LIST	.LIS
/LIBRARY	.L32
/ANALYSIS-DATA	.ANA

Suppose you have entered the following source text in the file ALPHA.BLI:

```
MODULE MYTEST =
BEGIN
REQUIRE 'CBLISS';
LIBRARY 'TBLISS';
...
END
ELUDOM
```

And you use the following command line to compile it:

```
$ BLISS ALPHA
```

Because the BLISS–command-line contains no qualifier requesting that a library file be produced, the output of the compilation is an object module. Therefore, the compiler chooses the list of default types associated with object module output and searches first for ALPHA.B32; then, not finding that file, the compiler searches for ALPHA.BLI, which it finds and compiles.

In processing the module MYTEST in that file, the compiler encounters the REQUIRE declaration for the file CBLISS. Because no file type for CBLISS is given, the compiler uses the list of default types for files in a REQUIRE declaration and searches for CBLISS.R32, then CBLISS.REQ, then CBLISS.B32, and then CBLISS.BLI. When the compiler processes the LIBRARY declaration, it uses the default type list associated with library declarations and searches first for TBLISS.L32.

1.3 Command-Line Qualifiers

Command-line qualifiers provide control over many aspects of the compilation. Valid command-line qualifiers and their functions are as follows:

- /CHECK qualifier—controls the level of semantic checking done during compilation
- /CROSS_REFERENCE qualifier—includes cross-reference information in output listing
- General qualifiers—sets a variant value and specifies code and debugging information

- /MACHINE-CODE-LIST qualifier—provides output listing information concerning the form of the object part
- /OPTIMIZE qualifier—supplies code optimization strategies and directions
- Output qualifiers—defines the types of output to be produced
- /SOURCE_LIST qualifier—provides output listing information concerning the form of the source part
- /TERMINAL qualifier—controls output produced on a terminal

1.3.1 /CHECK Qualifier

The /CHECK qualifier controls the level of semantic checking done during compilation. The qualifier allows all legal BLISS syntax to be examined for semantic irregularities. Some examples of the use of the /CHECK qualifier are as follows:

- To suppress field name checking on structure accesses if the data segment declaration has no field-attribute, use the /CHECK qualifier as follows:

```
$ BLISS/CHECK=NOFIELD ALPHA
```

- To check for the use of uninitialized storage, use the /CHECK qualifier as follows:

```
$ BLISS/CHECK=INITIAL ALPHA
```

1.3.1.1 Syntax

/CHECK qualifier syntax is as follows:

```
check qualifier        /CHECK= { (check-value , . . . ) }
                               { check-value            }

check-value            { FIELD        | NOFIELD      }
                       { INITIAL      | NOINITIAL    }
                       { OPTIMIZE     | NOOPTIMIZE   }
                       { REDECLARE    | NOREDECLARE  }
```

1.3.1.2 Defaults

In the absence of a specific choice of check-value, the following values are assumed by default:

FIELD
INITIAL
OPTIMIZE
NOREDECLARE

1.3.1.3 Semantics

The /CHECK qualifier indicates that one or more check-values follow. The check-values have the following meanings:

Check-Value	Meaning
FIELD	Do not suppress field name checking.
NOFIELD	If the data segment declaration has no field-attribute, suppress field name checking on the structure accesses.
INITIAL	Check for the use of uninitialized LOCAL, STACKLOCAL, and REGISTER data storage.
NOINITIAL	Do not check for the use of uninitialized storage.
OPTIMIZE	Check for suspicious optimizations—for example, constant folding expressions of a form that is always false, such as: $$.X < 0,8,1 > EQL$$
NOOPTIMIZE	Do not check for suspicious optimizations.
REDECLARE	Check for the redeclaration of a name within a nested scope.
NOREDECLARE	Do not check for the redeclaration of a name.

1.3.2 /CROSS_REFERENCE Qualifier

The /CROSS_REFERENCE qualifier allows a cross-reference listing to be included with the compiler listing or a cross-reference data file to be created, or both (refer to Section 1.3.5). Some examples of using the /CROSS_REFERENCE qualifier are as follows:

- To have a cross-reference listing included with the normal source compiler listing, use the /CROSS_REFERENCE qualifier in the command line as follows:

  ```
  $ BLISS/LIST/CROSS_REFERENCE ALPHA
  ```

  In addition to the object file, the compiler produces list file ALPHA.LIS, to which a cross-reference listing is appended.

- To have only a cross-reference listing produced (without the normal source compiler listing), use the following:

  ```
  $ BLISS/LIST/CROSS_REFERENCE/SOURCE=NOSOURCE -
  $_/MACH=NOOBJECT ALPHA
  ```

  In addition to the object file, the compiler produces list file ALPHA.LIS, which contains only cross-reference information.

- To produce a compiler listing with a cross-reference listing, use the following:

  ```
  $ BLISS/LIST/CROSS_REFERENCE ALPHA
  ```

  In addition to the object file, the compiler produces list file ALPHA. LIS, to which a cross reference listing is appended.

- To produce a listing with cross-references that include multiple references to the same type symbol occurring on the same source line, use the following:

  ```
  $ BLISS/LIST/CROSS_REFERENCE=(MULTIPLE) ALPHA
  ```

  In addition to the object file, the compiler produces list file ALPHA.LIS, to which a cross-reference listing is appended that includes multiple references to the same symbol.

1.3.2.1 Syntax

/CROSS_REFERENCE qualifier syntax is as follows:

reference-qualifier	{ /CROSS_REFERENCE { = reference-value } }
reference-value	{ MULTIPLE | NOMULTIPLE }

1.3.2.2 Defaults

In the absence of an explicit choice of reference value, the value NOMULTIPLE is assumed by default.

1.3.2.3 Semantics

The /CROSS_REFERENCE qualifier indicates that a reference-value can be given for the compilation. The reference-value has the following meanings:

Reference-Value	Meaning
MULTIPLE	Allow all multiple references (of the same reference type) to a symbol, occurring on the same source line, to be included in the cross-reference listing.
NOMULTIPLE	Exclude from the cross-reference listing all multiple references to a symbol occurring on the same source line.

1.3.3 General Qualifiers

General qualifiers are used to specify code and debugging information and to set the value for the lexical function %VARIANT. Some examples of the use of general qualifiers follow:

- To conserve object-file storage space, use the /NOTRACEBACK qualifier in the command line, as follows:

  ```
  $ BLISS/NOTRACEBACK ALPHA
  ```

  The compiler produces the minimum size object module in ALPHA.OBJ by omitting all debugging and traceback information.

- To include the necessary debugging information in the object module so that you can symbolically address declarations other than routine declarations, use the /DEBUG qualifier, as follows:

  ```
  $ BLISS/DEBUG ALPHA
  ```

  The compiler reads the source from ALPHA.B32 and creates an object file ALPHA.OBJ, which includes additional debug tables.

- To check the syntax of a program you do not intend to execute, use the /NOCODE qualifier to save compilation time, as follows:

  ```
  $ BLISS/LIST/NOCODE ALPHA
  ```

- To set the value of the lexical function %VARIANT to 17, for example, use the /VARIANT qualifier as follows:

  ```
  $ BLISS/VARIANT=17 ALPHA
  ```

1.3.3.1 Syntax

General qualifier syntax is as follows:

```
                    { /TRACEBACK              | /NOTRACEBACK }
                    { /DEBUG                  | /NODEBUG     }
general qualifier   { /CODE                   | /NOCODE      }
                    { /VARIANT { =value }     |              }
                    { /ERROR_LIMIT { =value } |              }
```

If the /NOTRACEBACK qualifier is given, then the /DEBUG qualifier is meaningless and, therefore, should not be given.

1.3.3.2 Defaults

In the absence of an explicit choice of general qualifier, the following qualifiers are assumed by default:

/TRACEBACK
/NODEBUG
/CODE
/VARIANT=0
/ERROR_LIMIT=30

The compiler produces code, does not include the additional debugging information in the object file, and sets the value of %VARIANT to zero.

If the general qualifier /VARIANT is given without a specified value, then a value of 1 is assumed.

1.3.3.3 Semantics

The command qualifiers have the following meanings:

/TRACEBACK	Generates information in the object module that can be used by the VMS Debugger to locate module, routine, and program section names.
/NOTRACEBACK	Produces the minimum size object module. Does not include any information for debugging or tracing.
/DEBUG	Generates information in the object module that can be used by the VMS Debugger to reference names declared within the BLISS module.
/NODEBUG	Does not generate any additional debugging information. If this qualifier is applied either explicitly or by default, the VMS Debugger can only locate module, routine, and PSECT names.
/CODE	Generates object code for the BLISS source module.
/NOCODE	Performs only a syntax check of the program.
/VARIANT	Sets %VARIANT to 1.
/VARIANT=n	Sets %VARIANT to n, where n is a decimal integer in the range: $-(2**31) \leq n \leq (2**31)-1$
/ERROR_LIMIT	Sets limit to 1.
/ERROR_LIMIT=n	Terminates compilation after n error level diagnostics are encountered.

1.3.3.4 Discussion

An object module can be produced with the following degrees of information for the linker, debugger, and the operating system:

- No information (/NOTRACEBACK)
- Basic information about modules, routines, and program sections (/TRACEBACK and /NODEBUG)
- Information about modules, routines, program sections, and data segment names (/TRACEBACK and /DEBUG)

The default object module contains the basic information. For example, if a program fails because of an access violation, VMS can display the state of the program when the violation occurred through its traceback facility. Further, you can use the VMS Debugger to refer to modules, routines, and PSECTs symbolically and to call routines.

The /NOTRACEBACK qualifier should be used only when object modules for well-checked-out programs are being generated and when the space to be occupied by these modules must be kept at a minimum. Using the /NOTRACEBACK qualifier reduces the size of the object module and, to a lesser degree, the size of the executable image, but deprives the object module of information that could be valuable at execution time.

1.3.4 /MACHINE_CODE_LIST Qualifier

The /MACHINE_CODE_LIST qualifier is used to give the compiler instructions about the form of the object part of the output listing. Some examples of using the /MACHINE_CODE_LIST qualifier are as follows:

- To obtain an output listing that can be subsequently edited and then reassembled by the VAX MACRO assembler, use the ASSEMBLER code-value, as follows:

  ```
  $ BLISS/LIST/MACHINE_CODE_LIST=ASSEMBLER ALPHA
  ```

- To obtain a listing that can be assembled and that does not contain binary, include the NOBINARY code-value:

  ```
  $ BLISS/LIST/MACHINE_CODE_LIST=(ASSEMBLER, NOBINARY) ALPHA
  ```

The form of the output listing is described in Section 2.2. The object code part of that listing depends on the machine-code-list qualifier.

1.3.4.1 Syntax

/MACHINE_CODE_LIST qualifier syntax is as follows:

```
qualifier /MACHINE_CODE_LIST= { ( code-value , . . . ) }
                              {   code-value            }

              { OBJECT        | NOOBJECT        }
              { ASSEMBLER     | NOASSEMBLER     }
code-value    { SYMBOLIC      | NOSYMBOLIC      }
              { BINARY        | NOBINARY        }
              { COMMENTARY    | NOCOMMENTARY    }
              { UNIQUE_NAMES  | NOUNIQUE_NAMES  }
```

1.3.4.2 Defaults

In the absence of an explicit choice of code-value, the following values are assumed by default:

OBJECT
NOASSEMBLER
SYMBOLIC
BINARY
COMMENTARY
NOUNIQUE_NAMES

The compiler produces a listing that resembles the output listing of the VAX MACRO assembler.

1.3.4.3 Semantics

The /MACHINE_CODE_LIST qualifier indicate that one or more code-values follow. The code-values have the following meanings:

Code-Value	Meaning
OBJECT	Produce the object part of the output listing.
NOOBJECT	Suppress the object part of the output listing.
ASSEMBLER	Produce a listing that can be assembled, by listing the assembler instructions produced as a result of compiling the BLISS program and including all other information within comments. If this output is to be assembled, the qualifiers /SOURCE_LIST=NOHEADER and /MACHINE_CODE_LIST=UNIQUE_NAMES must also be specified.
NOASSEMBLER	Do not list the assembler instructions.
SYMBOLIC	Include a machine code listing that uses names from the BLISS source program.
NOSYMBOLIC	Do not include a machine code listing that uses source program names.
COMMENTARY	Include a machine-generated commentary in the object code listing. At this time, the machine-generated commentary is limited to a cross-reference.
NOCOMMENTARY	Do not include a commentary field in the object code listing.
BINARY	Include a listing of the binary for each instruction in the object code listing.
NOBINARY	Do not include a listing of the binary
UNIQUE_NAMES	Replace names by machine-generated names so that all names are unique, regardless of scope, so that the resulting listing can be correctly assembled. (See ASSEMBLER above.)
NOUNIQUE_NAMES	Do not replace names by unique names.

Each of the code-values is described and illustrated in Section 2.2.3 in connection with the discussion of the output compilation.

1.3.5 /OPTIMIZE Qualifier

The /OPTIMIZE qualifier is used to supply directions to the compiler about the degree and type of optimization wanted, and to make assertions about the program so that the compiler can select the appropriate optimization strategies. Some examples of the use of the /OPTIMIZE qualifier are as follows:

- To increase the compilation speed by omitting some standard optimizations, use either the /QUICK qualifier or the /OPTIMIZE qualifier with the value QUICK in the command line, as follows:

  ```
  $ BLISS/OPTIMIZE=QUICK ALPHA
  ```

  Note that the use of either the QUICK value or the /QUICK qualifier turns off flow analysis. (Refer to Section 7.1.2.)

- To get minimum optimization, use the /OPTIMIZE qualifier with the value LEVEL:0, as follows:

  ```
  $ BLISS/OPTIMIZE=LEVEL:0 ALPHA
  ```

- To obtain maximum optimization, use the /OPTIMIZE qualifier with the value LEVEL:3, as follows:

  ```
  $ BLISS/OPTIMIZE=LEVEL:3 ALPHA
  ```

- To direct the compiler to use techniques that may generate a larger program in order to increase its operating efficiency, give the /OPTIMIZE qualifier with the value SPEED, as follows:

  ```
  $ BLISS/OPTIMIZE=SPEED ALPHA
  ```

- To inform the compiler that the program uses pointers to manipulate named data, use the /OPTIMIZE qualifier with the value NOSAFE, as follows:

  ```
  $ BLISS/OPTIMIZE=NOSAFE ALPHA
  ```

A detailed discussion of the optimizations resulting from using the /OPTIMIZE qualifier is given in Chapter 7.

1.3.5.1 Syntax

/OPTIMIZE qualifier syntax is as follows:

```
optimize qualifier        /OPTIMIZE= { ( optimize-value , . . . ) }
                                     { optimize-value              }

optimize-value            { QUICK                    | NOQUICK }
                          { SPEED                    | SPACE   }
                          { LEVEL : optimize-level             }
                          { SAFE                     | NOSAFE  }

optimize-level            { 0 | 1 | 2 | 3 }
```

The optimize-values SPEED and SPACE are mutually exclusive; they must not be given in the same command line.

1.3.5.2 Defaults

In the absence of an explicit choice of optimize-value, the following values are assumed by default:

NOQUICK
SPACE
LEVEL:2
SAFE

The compiler is directed to do the following:

- Perform normal optimization, biasing the speed/space trade-off in favor of minimum program size
- Assume that all variables are addressed by name only
- Perform optimization across mark points (see Section 7.1.2.2.)

1.3.5.3 Semantics

The /OPTIMIZE qualifier indicates that one or more optimize-values are given. Optimize-values have the following meanings.

Optimize-Value	Meaning
QUICK	Omit some standard optimizations (for example, turn off flow analysis) in order to increase compilation speed.
NOQUICK	Perform standard optimizations.
SPEED	Increase the potential execution speed of the program being compiled (if possible) by using more space where necessary. For more information on the effect of this value, see Section 7.1.4. Note that SPEED is equivalent to the module-switch ZIP.
SPACE	Keep program size to a minimum at the possible expense of operating speed. For more information on the effect of this value, see Section 7.1.4. Note that SPACE is equivalent to the module-switch NOZIP.
LEVEL	Optimize the program being compiled according to the optimize-level given, as follows: **Optimize-Level** / **Meaning** 0 — Minimum optimization 1 — Subnormal optimization 2 — Normal optimization 3 — Maximum optimization LEVEL:3 optimizes speed at the expense of space in the same way as SPEED. For more information on the effect of this value, see Section 7.2.
SAFE	Assume that all named data-segments are referenced by name only and not manipulated indirectly in any way, and use optimization techniques that exploit this fact. For more information on the effect of this value, see Section 7.1.2.1.
NOSAFE	Assume that sometimes a named data-segment is referenced by means of a computed expression and, therefore, some optimization techniques cannot be used.

1.3.6 Output Qualifiers

Output qualifiers are used to indicate the type of output to be produced from a BLISS–32 compilation and to give names for the files to be produced when you do not want to use the default names. The following are examples of output qualifiers:

- To suppress the production of an object file, use the /NOOBJECT qualifier in the command line, as follows:

  ```
  $ BLISS/NOOBJECT ALPHA
  ```

  The BLISS–32 compiler reads the source in the file ALPHA.B32 and produces no output files. The only outputs are the error messages and summary information produced at the terminal.

- To obtain a list file for a single source file, use the /LIST qualifier, as follows:

  ```
  $ BLISS/LIST ALPHA
  ```

  The BLISS–32 compiler produces an object file ALPHA.OBJ and a list file ALPHA.LIS. (The /LIST qualifier is assumed by default in a batch command.)

- To use a different name for the object or list files, use the following qualifiers:

  ```
  $ BLISS/OBJECT=BETA/LIST=GAMMA ALPHA
  ```

  The compiler reads the input file ALPHA.B32 and produces the object file BETA.OBJ and the list file GAMMA.LIS.

- To produce an analysis data file for a VAX Source Code Analyzer (SCA) library, use the following qualifier:

  ```
  $ BLISS/ANALYSIS_DATA=SCALIB ALPHA
  ```

  The compiler reads the input file ALPHA.B32 and produces the analysis data file SCALIB.ANA.

- To produce a library file rather than an object file, use the /LIBRARY qualifier, as follows:

  ```
  $ BLISS/LIBRARY ALPHA
  ```

  The compiler reads the input file ALPHA.B32 and produces the library file ALPHA.L32.

1.3.6.1 Syntax

Output qualifier syntax is as follows:

```
output        { /OBJECT {=file-spec}          | /NOOBJECT        }
qualifier     { /LIST {=file-spec}            | /NOLIST          }
              { /LIBRARY {=file-spec}         | /NOLIBRARY       }
              { /ANALYSIS_DATA {=file-spec}   | /NOANALYSIS_DATA }
```

The compiler can produce either a library or an object file, but not both. Therefore, the file-designators /OBJECT and /LIBRARY are mutually exclusive; they must not be given in the same command line.

1.3.6.2 Defaults

In interactive mode, in the absence of an explicit choice of output qualifier, the following qualifiers are assumed by default:

/OBJECT
/NOLIST
/NOLIBRARY
/NOANALYSIS_DATA

In batch mode, a list file is produced by default; thus, the following qualifiers are assumed:

/OBJECT
/LIST
/NOLIBRARY
/NOANALYSIS_DATA

1.3.6.3 Semantics

The output qualifiers have the following meanings:

/OBJECT=file-spec	Produce an object file in the file specified by file-spec.
/OBJECT	Produce an object file in the file specified by 'input-file-name.OBJ'.
/NOOBJECT	Do not produce an object file.
/LIST=file-spec	Produce a list file in the file specified by file-spec.
/LIST	Produce a list file in the file specified by 'input-file-name.LIS'.
/NOLIST	Do not produce a list file.
/LIBRARY=file-spec	Produce a library file in the file specified by file-spec.
/LIBRARY	Produce a library file in the file specified by 'input-file-name.L32'.
/NOLIBRARY	Do not produce a library file.
/ANALYSIS_DATA=file-spec	Produce an analysis data file in the file specified by file-spec.
/ANALYSIS_DATA	Produce an analysis data file in the file specified by 'input-file-name.ANA'.
/NOANALYSIS_DATA	Do not produce an analysis data file.

1.3.7 /SOURCE_LIST Qualifier

The /SOURCE_LIST qualifier is used to supply information about the form of the source part of the output listing. Some examples of using the /SOURCE_LIST qualifier are as follows:

- To obtain a paged listing with 44 lines on each page, give the following /SOURCE_LIST qualifier:

```
$ BLISS/LIST/SOURCE_LIST=PAGE_SIZE:44 ALPHA
```

- To obtain an unpaged listing in which the macro expansions are given, use the following source-value:

```
$ BLISS/LIST/SOURCE_LIST=(NOHEADER,EXPAND MACROS) ALPHA
```

- To obtain a listing that contains the contents of the REQUIRE files given in REQUIRE declarations, use the following source-value:

```
$ BLISS/LIST/SOURCE_LIST=REQUIRE ALPHA
```

1.3.7.1 Syntax

/SOURCE_LIST qualifier syntax is as follows:

```
source-list qualifier     /SOURCE_LIST= { ( source-value , . . . ) }
                                        { source-value             }

                          { HEADER            | NOHEADER         }
                          { PAGE_SIZE :      number-of-lines     }
                          { LIBRARY           | NOLIBRARY        }
source-value              { REQUIRE           | NOREQUIRE        }
                          { EXPAND_MACROS     | NOEXPAND_MACROS  }
                          { TRACE_MACROS      | NOTRACE_MACROS   }
                          { SOURCE            | NOSOURCE         }

number-of-lines           { 20 | 21 | 22 | . . . }
```

1.3.7.2 Defaults

In the absence of an explicit choice of source-value, the following values are assumed by default:

HEADER
PAGE_SIZE:58
NOLIBRARY
NOREQUIRE
NOEXPAND_MACROS
NOTRACE_MACROS
SOURCE

The compiler produces a paged listing with 58 lines on each page, in which no expansion or tracing is included.

1.3.7.3 Semantics

The /SOURCE_LIST qualifier indicates that one or more source-values are given for the compilation. The source-values have the following meanings:

Source-Value	Meaning
HEADER	Page the listing produced on the list file and include a heading on each page.
NOHEADER	Do not page the listing, do not include headings, and do not produce statistics in the compilation summary. Use this value if the listing is to be assembled.
PAGE_SIZE:lines	Use the number of lines specified for each page of the list file. The number of lines must be greater than 19.
LIBRARY	Produce a trace in the listing file identifying the library after a LIBRARY declaration and the first use of each name whose definition is obtained from a library file. For an example of a library trace, see Section 2.2.4.2.
NOLIBRARY	Do not produce a trace identifying any libraries and their contributions.
REQUIRE	Include source text obtained from require files in listing.
NOREQUIRE	No not include require-file text.

Source-Value	Meaning
EXPAND_MACROS	Include the expansion of each macro call in the listing file. For an example of a macro expansion, see Section 2.2.4.3.
NOEXPAND_MACROS	Do not include the expansion of macros.
TRACE_MACROS	Include a trace of each macro expansion. That is, include the parameter binding and any intermediate forms of expansion, as well as the result of the expansion. For an example of a macro trace, see Section 2.2.4.4.
NOTRACE_MACROS	Do not include a trace of macro expansions.
SOURCE	Increment the listing control counter. Output is listed when the listing control counter is positive and not listed when the counter is zero or negative.
NOSOURCE	Decrement the listing control counter.

1.3.8 /TERMINAL Qualifier

The /TERMINAL qualifier is used to control the output that is sent to the terminal. You can have errors or statistics printed or suppressed on the terminal during the compilation of a BLISS program. Some examples of the use of the /TERMINAL qualifier are as follows:

- To see the statistics for each routine as they are produced during the compilation, specify the terminal qualifier as follows:

```
$ BLISS/TERMINAL=STATISTICS ALPHA
```

- To suppress error messages and obtain statistics, use the following:

```
$ BLISS/TERMINAL=(NOERRORS,STATISTICS) ALPHA
```

1.3.8.1 Syntax

/TERMINAL qualifier syntax is as follows:

terminal qualifier	/TERMINAL= { (terminal-value, . . .) | terminal-value }
terminal-value	{ ERRORS | NOERRORS STATISTICS | NOSTATISTICS }

1.3.8.2 Defaults

In the absence of an explicit choice of terminal-value, the following values are assumed by default:

ERRORS
NOSTATISTICS

Errors are reported on the terminal during the compilation, but statistics are suppressed.

1.3.8.3 Semantics

The /TERMINAL qualifier indicates that one or more terminal-values follow. The terminal-values have the following meanings:

Terminal-Value	Meaning
ERRORS	List each error on the terminal as it is encountered in the compilation.
NOERRORS	Do not list errors on the terminal.
STATISTICS	List the name and size of each routine on the terminal after each routine is compiled.
NOSTATISTICS	Do not list routine names and sizes.

1.3.9 Qualifier Names Versus Switch Names

Some directions can be given to the compiler either by command line qualifiers or by switch settings contained in the module being compiled. In some cases, the qualifier name is the same as the switch name (module switches and SWITCHES declarations), while in other cases it is similar but not identical. The names of the corresponding qualifiers and switch items are given in Table Table 1–1.

Table 1–1 Correspondence Between Qualifier and Switch Names

Qualifier Name	Module-Head Switch	SWITCHES-Declaration Switch
/CODE	CODE	n/a
/DEBUG	DEBUG	n/a
/MACHINE_CODE_LIST=ASSEMBLER	LIST(ASSEMBLY)	LIST(ASSEMBLY)
/MACHINE_CODE_LIST=BINARY	LIST(BINARY)	LIST(BINARY)
/MACHINE_CODE_LIST=COMMENTARY	LIST(COMMENTARY)	LIST(COMMENTARY)
/MACHINE_CODE_LIST=OBJECT	LIST(OBJECT)	LIST(OBJECT)
/MACHINE_CODE_LIST=SYMBOLIC	LIST(SYMBOLIC)	LIST(SYMBOLIC)
/MACHINE_CODE_LIST=UNIQUE_NAMES	UNAMES	UNAMES
/OPTIMIZE=LEVEL:n	OPLEVEL=n	n/a
/OPTIMIZE=SAFE	SAFE	SAFE
/OPTIMIZE=SPACE	NOZIP	NOZIP
/OPTIMIZE=SPEED	ZIP	ZIP
/SOURCE_LIST=EXPAND_MACROS	LIST(EXPAND)	LIST(EXPAND)
/SOURCE_LIST=LIBRARY	LIST(LIBRARY)	LIST(LIBRARY)
/SOURCE_LIST=REQUIRE	LIST(REQUIRE)	LIST(REQUIRE)
/SOURCE_LIST=SOURCE	LIST(SOURCE)	LIST(SOURCE)
/SOURCE_LIST=TRACE_MACROS	LIST(TRACE)	LIST(TRACE)
/TERMINAL=ERRORS	ERRS	ERRS

1.3.10 Qualifiers and Default Settings

Qualifiers given in the command line alter the default settings assumed for module-head switches. A switch setting given in the module head overrides the corresponding qualifier given in the command-line; any switch setting given in a SWITCHES-declaration overrides the setting given in the module head.

Suppose you are compiling two modules. The first module ALPHA.B32 has a module switch CODE. The second module BETA.B32 has no switches. The BLISS command line is as follows:

```
$ BLISS/NOCODE ALPHA,BETA
```

The qualifier /NOCODE changes the initial default setting from /CODE to /NOCODE. When the module ALPHA.B32 is compiled, code is produced because ALPHA.B32 has the module-head switch CODE, which overrides the default setting. When the module BETA.B32 is compiled, no code is produced because it takes its setting of that switch from the initial default setting established in the command line.

1.3.11 Positive and Negative Forms of Qualifiers

In general, two forms of a qualifier are allowed: a positive form and a negative form. For example, /CODE (the positive form) directs the compiler to generate code, and /NOCODE (the negative form) directs the compiler to suppress code generation. Generally, positive and negative forms of a qualifier are mutually exclusive; however, exceptions can occur, such as in the following example:

```
$ BLISS/LIST ALPHA,BETA/NOLIST,GAMMA
```

The qualifier /LIST creates ALPHA.LIS and GAMMA.LIS, whereas the /NOLIST qualifier prevents the creation of a listing file for BETA.B32.

1.3.12 Abbreviations of Qualifier and Value Names

Command qualifier names and value names can be abbreviated. A valid abbreviation consists of the minimum number of characters required to identify a given command keyword without ambiguity. A list of the BLISS–32 command abbreviations and values is given in Appendix A.

2
Compiler Output

This chapter describes compiler output as it appears in terminal output, various list file formats, and error messages.

The input to a BLISS compilation is a BLISS module. As an example, consider the following module: it contains two OWN declarations and three ROUTINE declarations. The routine IFACT computes the factorial of its argument by an iterative method. The routine RFACT computes the factorial of its argument by a recursive method. The routine MAINPROG provides some test calls on IFACT and RFACT. Factorial routines are described in the *BLISS Language Reference Manual*.

```
module testfact (main = mainprog)
begin

own
    a,
    b;

routine ifact (n) =
    begin
    local
        result;
    result = 1;
    incr i from 2 to .n do
        result = .reult*.i;
    .result
    end;

routine rfact (n) =
    if .n gtr 1 then .n*rfact (.n - 1) else 1;

routine mainprog : novalue =
    begin
    a = ifact (5);
    b = rfact (5);

    1                             ! VMS wants a success return
    end;

end
eludom
```

This module is used in the following sections to illustrate various BLISS compilation output listings. Two coding errors (missing equal sign after the module-head and misspelled data-name) are included to illustrate the error reporting facility of BLISS.

2.1 Terminal Output

The compiler produces two kinds of information on the terminal: error messages and statistics. You can request or suppress each of these by using the /TERMINAL qualifier, as described in Section 1.3. By default, error messages are reported during compilation, but statistics are suppressed. A final compilation summary is evoked as part of a statistics request.

Error messages show the source program line associated with the error followed by a description of the error. The statistics show the name of each routine declaration in the module and the number of bytes associated with that declaration. The compilation statistics give the number of warning and error messages, the number of code bytes and data bytes used by the program, the run time and elapsed time required for the compilation, and the number of pages of VAX memory required for the compilation.

The last line of the terminal output indicates whether the compilation produced an object file or a library file. If an object file is produced, the last line is as follows:

```
;Compilation Complete
```

If a library file is produced, the last line is as follows:

```
;Library Precompilation Complete
```

Consider the terminal output for the sample module TESTFACT contained in the file MYPROG.B32. To obtain both kinds of information, compile the module by using the following BLISS command line:

```
$ BLISS/TERMINAL=STATISTICS MYPROG
```

The qualifier /TERMINAL=STATISTICS is used so that both types of output are sent to the terminal. The terminal output is as follows:

```
;   0002  0     begin
% WARN#048      1  L1:0002
  Syntax error in module head
;   0014   2           result = .reult*.I;
% WARN_#000     ..................1  L1:0014
  Undeclared name:  REULT
IFACT 22
RFACT 26
MAINPROG 28
; Information:  0
; Warnings:     2
; Errors:       0
; Size:         76 code + 8 data bytes
; Run time:        00:01.5
; Elapsed time:    00:03.5
; Lines/CPU Min:    1663
; Lexemes/CPU-Min:  6252
; Memory used:  15 pages
; Compilation Complete
%BLS32-W-ENDDIAGS, Compilation with 2 diagnostics
```

Terminal output for compiling MYPROG includes two warnings, which are described later. Statistics following the warnings show the number of bytes required for each routine. The module TESTFACT contains three routine declarations, namely, IFACT, RFACT, and MAINPROG, and they use 22, 26, and 28 bytes, respectively. The compilation summary shows that the compiling of TESTFACT required 1.2 seconds of processor time and that 3.0 seconds elapsed; moreover, 1663 source lines comprising 6252 lexemes were processed per

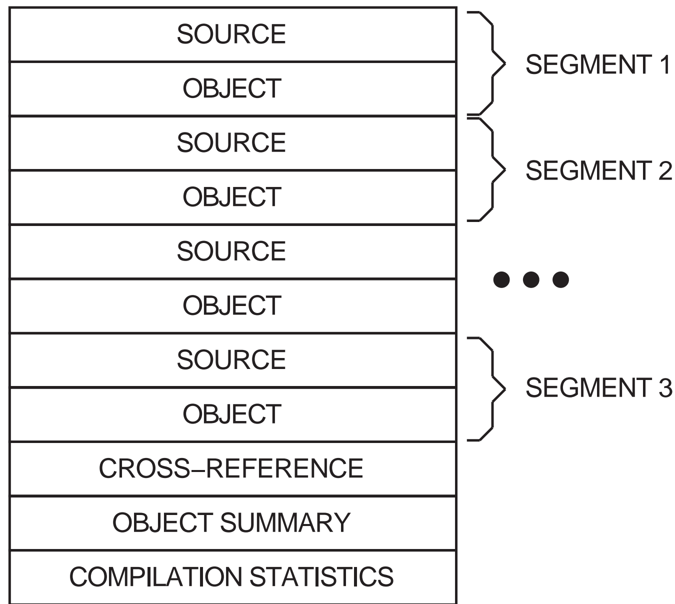
CPU minute. The compilation required 15 pages of memory, excluding memory required by the compiler itself.

2.2 Output Listing

The output listing produced as a result of a BLISS compilation consists of source listings, which include any error messages, object listings, and a compilation summary.

When the compiler completes the processing of a routine declaration, it produces the source and object listing for that declaration and any nonroutine declarations that preceded it. In this way, the output listing is divided into a sequence of segments. See Figure 2–1.

Figure 2–1 Compiler Output Listing Sequence



Both the source and object parts of a routine segment can be suppressed and the format of the object part can be changed by including switches in the module or qualifiers in the command line. In the absence of any explicit instruction, both source and object parts are produced. If the object part of the program is produced, an object summary is given. The object summary contains a PSECT summary and, if the compilation included any LIBRARY declarations, a summary of library usage. The compilation summary contains the same information as given in the compilation summary at the terminal.

The complete output listing for the module TESTFACT occupies several pages (refer to Section 2.2.3.1). The routine segment for the routine IFACT contains the module heading, the OWN declaration, and the routine declarations for IFACT. The following sections discuss each part of the output listing.

2.2.1 Listing Header

Listing headers consist of two lines; each line consists of three fields separated by at least one space. The first field contains information in print positions 1 through 15; the second extends from 17 through 63; the last extends from 65 through 132. The contents of each field are left-justified within the field. Figure 2–2 illustrates the listing header format.

Figure 2–2 Listing Header Format

PRINT POSITION	1 – 15	17 – 63	65 – 132
	NAME	TITLE	PROCESSOR IDENTIFICATION
	IDENT	SUBTITLE	SOURCE IDENTIFICATION

ZK–1369–GE

The name and ident fields contain the same information as that contained in the object file module headers. Some processors must generate the first page header before this information is available. Thus, the first page of a module may be blank; subsequent pages must include the information if it appears in the object module. If the module name exceeds 15 characters, the title field begins 8 columns further to the right.

The title and subtitle fields contain user-supplied information; they identify the purpose of the module and routine. User-defined title and subtitle entries that are too long are right truncated at column 63. If the language processor makes no provision for you to supply this information, the fields are ignored and the processor and source identifications start in column 17. If the language processor allows only one set of title information, the subtitle field is used for standard identification of the portion of the listing represented. When you update the title or subtitle information in the first line of the source page, the listing for that page will include the updated information.

The processor identification field contains the date and time of compilation (in the form dy-mon-year hh:mm:ss) and the full product name of the language processor. This field includes the release version number, with the edit number appended to it. The page listing number appears as the last entry in this field. This number increments by 1 for each listing page produced from a concatenated source file, that is, in the listing file.

The source identification field contains the date and time of creation or last modification of the source file being read at the start of this page. It also contains the resultant file name of this source file. It is a fully qualified name, including the actual version number. If the name is too long, the leftmost field is right truncated. The source file page number appears last, in parentheses, and is 1 greater than the number of page marks (form feeds) read from the source.

2.2.2 Source Listing

The source part of the output listing reproduces the input to the BLISS compilation with annotation supplied by the compiler. The compiler annotation includes a 16- or 24-character preface string that precedes each line of input, and error message lines that follow each line on which one or more errors are detected.

The basic difference in preface string length is due to the fact that the 24-character preface contains the editor's line sequence numbers, whereas the 16-character string does not.

The 16-character preface string has the following general form:

```
;byznnnnbnnbbbbb
```

The 24-character preface string has the following general form:

```
;xxxxxbbyznnnnbnnbbbbbbb
```

Table 2–1 describes preface string components of each string. (Asterisks denote the components and columns of the 24-character preface string.)

Table 2–1 Format of Preface String in Source Listing

<table>
<tr><th>Item</th><th>Column</th><th>Meaning</th></tr>
<tr><td>;</td><td>1</td><td>The comment character; used to comment out the source line so that the output listing can be assembled by the VAX–11 MACRO assembler.</td></tr>
<tr><td>xxxxx
or
b</td><td>2–6†
or
2</td><td>The line number, if the file contains line sequence numbers; otherwise, one blank column.</td></tr>
<tr><td>bb†</td><td>7–8†</td><td>Blanks.</td></tr>
<tr><td>y</td><td>9†
or
3</td><td>A code that indicates the lexical processing level of the compiler. The codes that can appear in this column are described below:
<table>
<tr><th>Code</th><th>Meaning</th></tr>
<tr><td>C</td><td>Embedded comment, that is, text within %(. . .)%</td></tr>
<tr><td>D</td><td>Default lexeme stream for a keyword macro formal</td></tr>
<tr><td>L</td><td>Parameter list of a lexical function</td></tr>
<tr><td>M</td><td>Body of a macro definition</td></tr>
<tr><td>P</td><td>Parameter list of a macro call</td></tr>
<tr><td>U</td><td>Source text that is discarded by an unsatisfied lexical condition</td></tr>
</table>
If more than one such code applies (for example, an embedded comment nested within a macro body), the “innermost” code is printed.</td></tr>
<tr><td>z</td><td>10†
or
4</td><td>If the line comes from a file specified in a REQUIRE declaration, the code “R”; otherwise, a blank.</td></tr>
</table>

†Components of the 24-character preface string

(continued on next page)

Table 2–1 (Cont.) Format of Preface String in Source Listing

Item	Column	Meaning
nnnn	11–14† or 5–8	The BLISS line sequence number, beginning with 0001; it is increased by 1 each time a source line is read. This line number is referenced by error messages and by the commentary field of the object code listing. It is always incremented for source lines read from REQUIRE files, even though those lines may not be listed.
b	15† or 9	Blank.
nn	16–17† or 10–11	The begin-end block level number reflecting the depth of the code within each block structure.
bbbbbbb†	18–24†	Blanks.
bbbbb	12–16	Blanks.

†Components of the 24-character preface string

For example, consider the following line of the source input:

```
RESULT = 1;
```

If the above expression were the fourteenth line of the compilation, for example, the output listing for that line would be as follows:

```
;   0014  2     RESULT = 1;
```

The line number 0014 is the line assigned by the BLISS compiler, and the begin-end block depth number 2 indicates that the line of code occurs in the second block level. If the input line had an editor line sequence number 2300, the output listing for the line would be as follows:

```
;02300    0014  2       RESULT = 1;
```

If the input line comes from a REQUIRE file, the output listing includes an R, as follows:

```
;  R0014  2     RESULT = 1;
```

If the input line is part of a macro declaration, the output listing includes an M, as follows:

```
; M 0014  2     RESULT = 1;
```

The y item in the preface string column 3 (9 for a 24-character preface) is useful for detecting lexical errors. For example, if you forget to terminate a macro declaration, all the following lines in the program are then assumed to be part of that macro declaration and the error is not detected until the end of the program. You can find the beginning of the unterminated macro, however, by finding the point at which the M code first appears in the y field.

The following is an example of the source listing (the first segment of the module TESTFACT) that uses the 24-character preface string:

```
    .
    . (header)
    .
;00100    0001  0       module testfact (main = mainprog)
;00200    0002  0       begin
; WARN#048              1  L1:0002
; Syntax error in module head
;00300    0003  1
;00400    0004  1       own
;00500    0005  1           a,
;00600    0006  1           b;
;00700    0007
;00800    0008  1       routine ifact (n) =
;00900    0009  2           begin
;01000    0010  2           local
;01100    0011  2               result;
;01200    0012  2           result = 1;
;01300    0013  2           incr i from 2 to .n do
;01400    0014  2               result = .reult*.i;
; WARN#000              ..................1  L1:0014
; Undeclared name:  REULT
;01500    0015  2           .result
;01600    0016  1           end;
    .
    .
    .
```

Following three heading lines, the source of the module TESTFACT is reproduced. The 24-character preface string begins with a semicolon (;). Columns 7 and 8 are always blank, and columns 9 and 10 are blank because the lexical processing level is normal and the material is not from a REQUIRE file. Line numbers generated by the compiler occupy columns 11 through 14. Column 15 is always blank, and in columns 16 and 17 the begin-end block level count appears.

Two error messages are reported as part of the source listing. Section 2.5 discusses error messages and their meaning.

2.2.3 Object Listing

The object part of the listing has four possible fields: assembler input, assembler output, binary, and commentary. The parts of the object listing that are produced depend on the choice of machine-code-list code-values specified in the command line. Each part of the object listing has associated with it a machine-code-list code-value that allows it to be either printed or suppressed.

However, although 32 different forms of listing are theoretically possible, in practice only a few combinations of code-values are meaningful. The basic distinction among object listings is whether the listing replicates assembler input or assembler output. If the ASSEMBLER code-value is given, the object part is formatted so that it can be read by the assembler. If the NOASSEMBLER code-value is given, the object part is formatted to resemble assembler output.

The following combinations of the machine-code-list code-values are acceptable:

ASSEMBLER { SYMBOLIC | NOSYMBOLIC } COMMENTARY { BINARY | NOBINARY } { UNIQUE_NAMES | NOUNIQUE_NAMES }

NOASSEMBLER SYMBOLIC COMMENTARY { BINARY | NOBINARY } NOUNIQUE_NAMES

The commentary field requires little space and provides useful information so that you have no real need for the NOCOMMENTARY qualifier.

The question of whether to have the binary field appear on the listing is a question of personal preference. However, it may at times be useful for debugging purposes. If the binary field is omitted, the listing is likely to be more compressed, since additional operands can then be placed on the same line.

The compiler produces the following information for each field:

Field	Contents
ASSEMBLER field	Instructions in assembler form, for example: `MOVL   #1,R0`
SYMBOLIC field	Instructions in assembler form but using symbolic source names, for example: `MOVL   #1,RESULT`
BINARY field	Hexadecimal equivalent of instructions and data to enable easier debugging. The hexadecimal instructions appear in the same format as that produced by the VAX MACRO assembler as far as possible. The rightmost numeric field in the binary listing is the location counter, relative to the starting routine's address. This simplifies interaction with the VMS Debugger when you are setting breakpoints or examining instructions (refer to Section 3.3). The following codes are included in the hexadecimal information in the binary field to provide information about relocation of quantities:

Code	Meaning
Blank	Absolute quantity (no linker action).
V	Forward relocatable.
*	Complex.
'	Relocated (relative to a program section other than the code program section).

Code	Meaning
G	Either general addressing (position independent) or globally relocatable.
COMMENTARY field	A cross-reference to the source program line generating the code. If a program line generates more than one instruction line, commentary fields in the lines following the first generated instruction remain blank.

Some examples of the object part of the routine segment IFACT follow.

2.2.3.1 Default Object Listing

The listing in Example 2–1 was produced by the following command line:

```
$ BLISS/LIST MYPROG
```

In the listing, the binary field appears first, followed by the symbolic field, and then by the commentary field.

2.2.3.2 Assembler Input Listing

The listing in Example 2–2 was produced by compiling module TESTFACT with the following code-options:

```
$ BLISS/LIST/MACHINE_CODE_LIST:(ASSEMBLER,NOBINARY) MYPROG
```

In the listing, the assembler field appears first, followed by the symbolic field, and then by the commentary field. Observe that when both the assembler and symbolic fields are present, only the operands are given in the symbolic field to conserve space. Labels, instruction names, and assembler directives are not repeated.

Example 2–1 Default Object Listing

```
TESTFACT   2-Mar-1987 14:11:48  VAX BLISS--32 T4.3-793              Page   1
          22-Jul-1983 16:06:56  BLISS$:[ELLIS.B32]MYPROG.B32;13         (1)

;   0001  0     MODULE TESTFACT (MAIN = MAINPROG)
;   0002  0     BEGIN
; WARN#048      1  L1:0002
; Syntax error in module head
;   0003  1
;   0004  1     OWN
;   0005  1         A,
;   0006  1         B;
;   0007  1
;   0008  1     ROUTINE IFACT (N) =
;   0009  2         BEGIN
;   0010  2         LOCAL
;   0011  2             RESULT;
;   0012  2         RESULT = 1;
;   0013  2         INCR I FROM 2 TO .N DO
;   0014  2             RESULT = .REULT*.I;
; WARN#000      ..................1  L1:0014
; Undeclared name:  REULT
;   0015  2         .RESULT
;   0016  1         END;

                                      .TITLE  TESTFACT

                                      .PSECT  $OWN$,NOEXE,2

                      00000 A:        .BLKB   4
                      00004 B:        .BLKB   4

                                      .EXTRN  REULT

                                      .PSECT  $CODE$,NOWRT,2

                 0000 00000 IFACT:    .WORD   Save nothing              ; 0008
         50    01  D0 00002           MOVL    #1, RESULT                ; 0012
         51    01  D0 00005           MOVL    #1, I                     ; 0013
               06  11 00008           BRB     2$                        ;
50 0000G CF    51  C5 0000A 1$:       MULL3   I, REULT, RESULT          ; 0014
F5       51 04 AC  F3 00010 2$:       AOBLEQ  N, I, 1$                  ;
                   04 00015           RET                               ; 0016

; Routine Size:  22 bytes,    Routine Base:  $CODE$ + 0000
```

(continued on next page)

Example 2–1 (Cont.) Default Object Listing

```
;   0017  1
;   0018  1     ROUTINE RFACT (N) =
;   0019  1         IF .N GTR 1
;   0020  1        THEN
;   0021  1             .N * RFACT(.N - 1)
;   0022  1         ELSE
;   0023  1             1;


                  0000 00000 RFACT:  .WORD   Save nothing                ; 0018

TESTFACT    2-Mar-1987 14:11:48 VAX BLISS--32 T4.3-793                  Page   2
           22-Jul-1983 16:06:56 BLISS$:[ELLIS.B32]MYPROG.B32;13            (1)

           01     04 AC  D1 00002         CMPL    N, #1                  ; 0019
                     0E  15 00006         BLEQ    1$                     ;
7E    04   AC        01  C3 00008         SUBL3   #1, N, -(SP)           ; 0021
      EF   AF        01  FB 0000D         CALLS   #1, RFACT              ;
           50     04 AC  C4 00011         MULL2   N, R0                  ;
                         04 00015         RET                            ;
           50        01  D0 00016 1$:     MOVL    #1, R0                 ; 0019
                         04 00019         RET                            ; 0023

; Routine Size:  26 bytes,    Routine Base:  $CODE$ + 0016

;   0024  1
;   0025  1     ROUTINE MAINPROG =
;   0026  2         BEGIN
;   0027  2         A = IFACT(5);
;   0028  2         B = RFACT(5);
;   0029  2
;   0030  2         1                          ! VMS wants a success return
;   0031  1         END;


                       0000 00000 MAINPROG:
                                          .WORD   Save nothing           ; 0025
                     05  DD 00002         PUSHL   #5                     ; 0027
      C8   AF        01  FB 00004         CALLS   #1, IFACT              ;
    0000’  CF        50  D0 00008         MOVL    R0, A                  ;
                     05  DD 0000D         PUSHL   #5                     ; 0028
      D3   AF        01  FB 0000F         CALLS   #1, RFACT              ;
    0000’  CF        50  D0 00013         MOVL    R0, B                  ;
           50        01  D0 00018         MOVL    #1, R0                 ; 0031
                         04 0001B         RET                            ;

; Routine Size:  28 bytes,    Routine Base:  $CODE$ + 0030

;   0032  1     END
;   0033  0     ELUDOM



;                               PSECT SUMMARY
;
;   Name    Bytes                    Attributes
;
;  $OWN$      8  NOVEC,  WRT,  RD ,NOEXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)
;  $CODE$    76  NOVEC,NOWRT,  RD ,  EXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)
```

(continued on next page)

Example 2–1 (Cont.) Default Object Listing

```
TESTFACT    2-Mar-1987 14:11:48 VAX BLISS--32 T4.3-793                 Page   3
           22-Jul-1983 16:06:56 BLISS$:[ELLIS.B32]MYPROG.B32;13            (1)

; Information:  0
; Warnings:     2
; Errors:       0


;                          COMMAND QUALIFIERS

;       BLISS/LIST MYPROG

; Size:        76 code + 8 data bytes
; Run Time:        00:01.4
; Elapsed Time:    00:02.9
; Lines/CPU Min:    1455
; Lexemes/CPU-Min:  5470
; Memory Used:  19 pages
; Compilation Complete
```

Example 2–2 Assembler Input Listing

```
TESTFACT     2-Mar-1987 13:54:04    VAX BLISS--32 T4.3-793             Page   1
            22-Jul-1983 16:06:56    BLISS$:[ELLIS.B32]MYPROG.B32;13        (1)

;   0001  0     MODULE TESTFACT (MAIN = MAINPROG)
;   0002  0     BEGIN
; WARN#048      1  L1:0002
; Syntax error in module head
;   0003  1
;   0004  1     OWN
;   0005  1         A,
;   0006  1         B;
;   0007  1
;   0008  1     ROUTINE IFACT (N) =
;   0009  2         BEGIN
;   0010  2         LOCAL
;   0011  2             RESULT;
;   0012  2         RESULT = 1;
;   0013  2         INCR I FROM 2 TO .N DO
;   0014  2             RESULT = .REULT*.I;
; WARN#000      ..................1  L1:0014
; Undeclared name:  REULT
;   0015  2         .RESULT
;   0016  1         END;

        .TITLE  TESTFACT

        .PSECT  $OWN$,NOEXE,2

A:      .BLKB   4
B:      .BLKB   4

        .EXTRN  REULT

        .PSECT  $CODE$,NOWRT,2
```

(continued on next page)

Example 2–2 (Cont.) Assembler Input Listing

```
IFACT:  .WORD   ^M<>                             ;Save nothing          ; 0008
        MOVL    #1, R0                           ;#1, RESULT            ; 0012
        MOVL    #1, R1                           ;#1, I                 ; 0013
        BRB     2$                               ;2$                    ;
1$:     MULL3   R1, W^REULT, R0                  ;I, REULT, RESULT      ; 0014
2$:     AOBLEQ  4(AP), R1, 1$                    ;N, I, 1$              ;
        RET                                      ;                      ; 0016

; Routine Size:  22 bytes,    Routine Base:  $CODE$ + 0000

;   0017  1
;   0018  1     ROUTINE RFACT (N) =
;   0019  1         IF .N GTR 1
;   0020  1        THEN
;   0021  1             .N * RFACT(.N - 1)
;   0022  1         ELSE
;   0023  1             1;


RFACT:  .WORD   ^M<>                               ;Save nothing        ; 0018

TESTFACT     2-Mar-1987 13:54:04    VAX BLISS--32 T4.3-793              Page   2
            22-Jul-1983 16:06:56    BLISS$:[ELLIS.B32]MYPROG.B32;13         (1)

        CMPL    4(AP), #1                          ;N, #1               ; 0019
        BLEQ    1$                                 ;1$                  ;
        SUBL3   #1, 4(AP), -(SP)                   ;#1, N, -(SP)        ; 0021
        CALLS   #1, B^RFACT                        ;#1, RFACT           ;
        MULL2   4(AP), R0                          ;N, R0               ;
        RET                                        ;                    ;
1$:     MOVL    #1, R0                             ;#1, R0              ; 0019
        RET                                        ;                    ; 0023

; Routine Size:  26 bytes,    Routine Base:  $CODE$ + 0016

;   0024  1
;   0025  1     ROUTINE MAINPROG =
;   0026  2         BEGIN
;   0027  2         A = IFACT(5);
;   0028  2         B = RFACT(5);
;   0029  2
;   0030  2         1                            ! VMS wants a success return
;   0031  1         END;



MAINPROG:
        .WORD   ^M<>                               ;Save nothing        ; 0025
        PUSHL   #5                                 ;#5                  ; 0027
        CALLS   #1, B^IFACT                        ;#1, IFACT           ;
        MOVL    R0, W^A                            ;R0, A               ;
        PUSHL   #5                                 ;#5                  ; 0028
        CALLS   #1, B^RFACT                        ;#1, RFACT           ;
        MOVL    R0, W^B                            ;R0, B               ;
        MOVL    #1, R0                             ;#1, R0              ; 0031
        RET                                        ;                    ;

; Routine Size:  28 bytes,    Routine Base:  $CODE$ + 0030

;   0032  1     END
;   0033  0     ELUDOM
```

(continued on next page)

Example 2–2 (Cont.) Assembler Input Listing

```
;                                       PSECT SUMMARY
;
;  Name       Bytes                          Attributes
;
;  $OWN$       8  NOVEC,  WRT,  RD ,NOEXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)
;  $CODE$     76  NOVEC,NOWRT,  RD ,  EXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)

TESTFACT     2-Mar-1987 13:54:04    VAX BLISS--32 T4.3-793              Page   3
            22-Jul-1983 16:06:56    BLISS$:[ELLIS.B32]MYPROG.B32;13         (1)

; Information:  0
; Warnings:     2
; Errors:       0


;                                        COMMAND QUALIFIERS

;       BLISS/LIST/MACHINE_CODE_LIST:(ASSEMBLER,NOBINARY) MYPROG

; Size:         76 code + 8 data bytes
; Run Time:        00:01.3
; Elapsed Time:    00:01.7
; Lines/CPU Min:    1571
; Lexemes/CPU-Min:  5904
; Memory Used:  18 pages
; Compilation Complete

        .END    MAINPROG
```

2.2.4 Source Part Options

The following sections contain more output listings to illustrate different options for the source part of the list file. To illustrate these different forms, the sample program TESTFACT has to be made more complex.

Suppose the testing of the same program TESTFACT is complete, source code errors contained in the preceding examples have been corrected, and the data on the relative performance of the two factorial routines has been obtained. The next step is the production of a new module TEST that uses the factorial routine to take combinations, according to the following formula for obtaining the number of combinations of m things taken n at a time:

$$\binom{m}{n} = \frac{m!}{(m-n)!n!}$$

First, enter the routine declarations for IFACT and RFACT into separate REQUIRE files, named IFACT and RFACT, respectively. The module TEST can then use either routine by including the appropriate REQUIRE declaration.

Next, write a macro (COMBN.R32) for obtaining the combinations:

```
MACRO
     COMBINATIONS(N,M) =
          (IF N LSS M
          THEN ERROR()
          ELSE COMB(N,M)) %,

     COMB(M,N) =
          FACT(N)/(FACT(N-M)_*FACT(M)) %;
```

Then, precompile the macro declaration into a LIBRARY file as follows (include a LIBRARY declaration in the module TEST):

```
$ BLISS/LIBRARY COMBN
```

Finally, include some test combinations.

The following sections illustrate the different output listings obtained for that module by varying the command qualifiers.

2.2.4.1 Default Source Listing

The following command line generated the output listing in Example 2–3 for the module TEST:

```
$ BLISS/LIST/NOCODE TEST
```

Observe that, although the contents of the REQUIRE file are not printed, the lines within the file are numbered by the compiler. The output listing shows that lines 0011 through 0016 are used for this purpose.

2.2.4.2 Listing with LIBRARY and REQUIRE Information

The following command line generated the output listing in Example 2–4, which contains information from LIBRARY and REQUIRE files:

```
$ BLISS/NOCODE/LIST/SOURCE_LIST:(LIBRARY,REQUIRE) TEST
```

The LIBRARY file is identified following line 0009, and the first use of a name from that library is noted following line 0020. The contents of the REQUIRE file are given in lines 0011 through 0016.

2.2.4.3 Listing with Macro Expansions

The following command line generated the output listing in Example 2–5 to illustrate macro expansions, which follow lines 0020 and 0021:

```
$ BLISS/NOCODE/LIST/SOURCE_LIST:EXPAND_MACROS TEST
```

Observe that expansions are listed in the order in which they occur. The innermost expansion is printed first, followed by the outer expansion, which includes the expanded form of the inner macro. Therefore, the last line of the macro expansion is the fully expanded form.

2.2.4.4 Listing with Macro Tracing

The following command line produced the output listing in Example 2–6, which contains macro tracing and macro expansion information:

```
$ BLISS/NOCODE/LIST/SOURCE_LIST:TRACE_MACROS TEST
```

The macro trace gives information about parameter binding in addition to the expansion information.

Example 2–3 Default Source Listing

```
TEST         2-Mar-1987 13:59:39    VAX BLISS--32 T4.3-793              Page   1
            16-Apr-1983 20:02:35    BLISS$:[ELLIS.B32]TEST.BLI;1            (1)

;   0001  0     module test (main = mainprog) =
;   0002  1     begin
;   0003  1
;   0004  1     own
;   0005  1         a,
;   0006  1         b;
;   0007  1     external routine
;   0008  1         error;
;   0009  1     library 'combn' ;
;   0010  1     require 'rfact' ;
;   0017  1
;   0018  1     routine mainprog =
;   0019  2         begin
;   0020  2         a = combinations (3, 2);
;   0021  2         b = combinations (6, 4);
;   0022  2
;   0023  2         1                           ! VMS wants a success return
;   0024  1         end;
;   0025  1
;   0026  1     end
;   0027  0     eludom

;                               Library Statistics
;
;                               -------- Symbols --------      Pages  Processing
;       File                    Total    Loaded   Percent      Mapped    Time
;
; BLISS$:[ELLIS.B32]COMBN.L32;1   2         2       100         6       00:00.1


;                                       COMMAND QUALIFIERS

;       BLISS/LIST/NOCODE TEST

; Run Time:         00:00.6
; Elapsed Time:     00:00.9
; Lines/CPU Min:     2842
; Lexemes/CPU-Min: 21894
; Memory Used:  10 pages
; Compilation Complete
```

Example 2–4 Output Listing with LIBRARY and REQUIRE File Data

```
TEST         2-Mar-1987 14:02:41    VAX BLISS--32 T4.3-793              Page   1
            16-Apr-1983 20:02:35    BLISS$:[ELLIS.B32]TEST.BLI;1            (1)

;   0001  0     module test (main = mainprog) =
;   0002  1     begin
;   0003  1
;   0004  1     own
;   0005  1         a,
;   0006  1         b;
;   0007  1     external routine
;   0008  1         error;
;   0009  1     library 'combn' ;
;; Library file BLISS$:[ELLIS.B32]COMBN.L32;1 produced by VAX BLISS--32 T4.3-793
;; on  2-Mar-1987 13:58:38
;   0010  1     require 'rfact' ;

TEST         2-Mar-1987 14:02:41    VAX BLISS--32 T4.3-793              Page   2
            13-Apr-1983 10:27:34    BLISS$:[ELLIS.B32]RFACT.REQ;2           (1)

;  R0011  1     ROUTINE FACT (N) =
;  R0012  1         IF .N GTR 1
;  R0013  1         THEN
;  R0014  1             .N * FACT(.N - 1)
;  R0015  1         ELSE
;  R0016  1            1;

TEST         2-Mar-1987 14:02:41    VAX BLISS--32 T4.3-793              Page   3
            16-Apr-1983 20:02:35    BLISS$:[ELLIS.B32]TEST.BLI;1            (1)

;   0017  1
;   0018  1     routine mainprog =
;   0019  2         begin
;   0020  2         a = combinations (3, 2);
;; Loaded symbol COMBINATIONS from library BLISS$:[ELLIS.B32]COMBN.L32;1
;; Loaded symbol COMB from library BLISS$:[ELLIS.B32]COMBN.L32;1
;   0021  2         b = combinations (6, 4);
;   0022  2
;   0023  2         1                          ! VMS wants a success return
;   0024  1         end;
;   0025  1
;   0026  1     end
;   0027  0     eludom

;                               Library Statistics
;
;                               -------- Symbols --------    Pages    Processing
;       File                    Total    Loaded   Percent    Mapped   Time
;
; BLISS$:[ELLIS.B32]COMBN.L32;1  2         2       100         6       00:00.1


;                                      COMMAND QUALIFIERS

;       BLISS/NOCODE/LIST/SOURCE_LIST:(LIBRARY,REQUIRE) TEST

; Run Time:        00:00.7
; Elapsed Time:    00:01.4
; Lines/CPU Min:    2417
; Lexemes/CPU-Min: 18626
; Memory Used:  10 pages
; Compilation Complete
```

Example 2–5 Output Listing with Macro Expansion Data

```
TEST         2-Mar-1987 14:14:51    VAX BLISS--32 T4.3-793              Page   1
            16-Apr-1983 20:02:35    BLISS$:[ELLIS.B32]TEST.BLI;1            (1)

;   0001  0     module test (main = mainprog) =
;   0002  1     begin
;   0003  1
;   0004  1     own
;   0005  1         a,
;   0006  1         b;
;   0007  1     external routine
;   0008  1         error;
;   0009  1     library 'combn' ;
;   0010  1     require 'rfact' ;
;   0017  1
;   0018  1     routine mainprog =
;   0019  2         begin
;   0020  2         a = combinations (3, 2);
;;         [COMB]=  FACT ( 3 ) / ( FACT ( 3 - 2 ) * FACT ( 2 ) )
;;         [COMBINATIONS]=  ( IF 3 LSS 2 THEN ERROR ( )
                              ELSE FACT ( 3 ) / ( FACT ( 3 - 2 ) * FACT ( 2 ) ) )
;   0021  2         b = combinations (6, 4);
;;         [COMB]=  FACT ( 6 ) / ( FACT ( 6 - 4 ) * FACT ( 4 ) )
;;         [COMBINATIONS]=  ( IF 6 LSS 4 THEN ERROR ( )
                              ELSE FACT ( 6 ) / ( FACT ( 6 - 4 ) * FACT ( 4 ) ) )
;   0022  2
;   0023  2         1                          ! VMS wants a success return
;   0024  1         end;
;   0025  1
;   0026  1     end
;   0027  0     eludom

;                               Library Statistics
;
;                               -------- Symbols --------   Pages     Processing
;       File                    Total    Loaded   Percent   Mapped    Time
;
; BLISS$:[ELLIS.B32]COMBN.L32;1   2         2       100      6        00:00.1


;                                       COMMAND QUALIFIERS

;       BLISS/NOCODE/LIST/SOURCE_LIST:EXPAND_MACROS TEST

; Run Time:        00:00.7
; Elapsed Time:    00:01.5
; Lines/CPU Min:    2492
; Lexemes/CPU-Min: 19200
; Memory Used:  10 pages
; Compilation Complete
```

Example 2–6 Output Listing with Macro Expansion and Tracing Data

(continued on next page)

Example 2–6 (Cont.) Output Listing with Macro Expansion and Tracing Data

```
TEST         2-Mar-1987 14:15:39    VAX BLISS--32 T4.3-793             Page   1
            16-Apr-1983 20:02:35    BLISS$:[ELLIS.B32]TEST.BLI;1            (1)

;   0001  0     module test (main = mainprog) =
;   0002  1     begin
;   0003  1
;   0004  1     own
;   0005  1         a,
;   0006  1         b;
;   0007  1     external routine
;   0008  1         error;
;   0009  1     library 'combn' ;
;   0010  1     require 'rfact' ;
;   0017  1
;   0018  1     routine mainprog =
;   0019  2         begin
;   0020  2         a = combinations (3, 2);
;;    [COMBINATIONS]: Parameter binding
;;    [COMBINATIONS](1)=  3
;;    [COMBINATIONS](2)=  2
;;    [COMBINATIONS]: Expansion
;;        [COMB]: Parameter binding
;;        [COMB](1)=  3
;;        [COMB](2)=  2
;;        [COMB]: Expansion
;;        [COMB]=  FACT ( 3 ) / ( FACT ( 3 - 2 ) * FACT ( 2 ) )
;;    [COMBINATIONS]=  ( IF 3 LSS 2 THEN ERROR ( )
                         ELSE FACT ( 3 ) / ( FACT ( 3 - 2 ) * FACT ( 2 ) ) )
;   0021  2         b = combinations (6, 4);
;;    [COMBINATIONS]: Parameter binding
;;    [COMBINATIONS](1)=  6
;;    [COMBINATIONS](2)=  4
;;    [COMBINATIONS]: Expansion
;;        [COMB]: Parameter binding
;;        [COMB](1)=  6
;;        [COMB](2)=  4
;;        [COMB]: Expansion
;;        [COMB]=  FACT ( 6 ) / ( FACT ( 6 - 4 ) * FACT ( 4 ) )
;;    [COMBINATIONS]=  ( IF 6 LSS 4 THEN ERROR ( )
                         ELSE FACT ( 6 ) / ( FACT ( 6 - 4 ) * FACT ( 4 ) ) )
;   0022  2
;   0023  2         1                           ! VMS wants a success return
;   0024  1         end;
;   0025  1
;   0026  1     end
;   0027  0     eludom

;                               Library Statistics
;
;                               -------- Symbols --------   Pages    Processing
;       File                    Total    Loaded   Percent   Mapped   Time
;
; BLISS$:[ELLIS.B32]COMBN.L32;1  2         2       100       6       00:00.1
```

(continued on next page)

Example 2–6 (Cont.) Output Listing with Macro Expansion and Tracing Data

```
;                                       COMMAND QUALIFIERS

TEST         2-Mar-1987 14:15:39    VAX BLISS--32 T4.3-793              Page   2
            16-Apr-1983 20:02:35    BLISS$:[ELLIS.B32]TEST.BLI;1            (1)

;       BLISS/NOCODE/LIST/SOURCE_LIST:TRACE_MACROS TEST

; Run Time:         00:00.7
; Elapsed Time:     00:01.1
; Lines/CPU Min:     2250
; Lexemes/CPU-Min: 17333
; Memory Used:  10 pages
; Compilation Complete
```

2.3 Cross-Reference Listing

The cross-reference listing is an optional part of the output listing that is produced by the compiler on request. Cross-reference data are generated on a module basis; therefore, the reference information associated with a given module will appear as the last module-specific item in the listing file, before the compilation summary and before any subsequent module data.

2.3.1 Cross-Reference Header

The cross-reference header is separated from the output listing header by a blank line and subsequently appears on the first two lines of each page of the reference listing. The cross-reference header is as follows:

```
 Symbol           Type  Defined   Referenced ...
--------------- -------- ----- -----------------
```

2.3.2 Cross-Reference Entries

The reference entries listed under each header name are fixed-length fields that are separated by a single space.

Symbol Field

The symbol field is used to list the names of the different symbols. The length of the field is fixed at the length of the longest name in the module. A name appears just once in the symbol field, defining its initial recognition by the compiler as a declared symbol. If multiple symbols are declared with the same name, lines directly following the first appearance of the name are used; however, for each subsequent recognition the field is left blank. For example:

```
ALPHA   .  .  .
        .  .  .
GAMMA   .  .  .
```

Type Field

The type field describes the condition (such as LOCAL, BIND, or BUILTIN) under which the symbol-name was used when it was declared. The field is eight characters long; therefore, symbol type abbreviations are used. The symbol type abbreviations are listed in Table 2–2.

Table 2–2 Symbol Type Abbreviations

Symbol Type	Abbreviation
Bind	Bind
Bind routine	BindRout
Builtin	Builtin
Compiletime	Comptime
Enable	Enable
External	External
External literal	ExtLit
External register	ExtReg
External routine	ExtRout
Field	Field
Fieldset	Fieldset
Forward	Forward
Forward routine	ForwRout
Global	Global
Global bind	GlobBind
Global bind routine	GlBiRout
Global literal	GlobLit
Global register	GlobReg
Global routine	GlobRout
Keyword macro	KeyWMacr
Keyword macro formal	KeyWForm
Label	Label
Linkage	Linkage
Literal	Literal
Local	Local
Macro	Macro
Map	Map
Macro formal	MacrForm
Symbol without a declaration	NotDecl
Own	Own
Psect	Psect
Register	Register
Routine	Routine
Routine formal	RoutForm
Structure	Structur

(continued on next page)

Table 2–2 (Cont.) Symbol Type Abbreviations

Symbol Type	Abbreviation
Stacklocal	Stackloc
Structure formal	StruForm
Name which is not bound	Unbound
Undeclare	Undeclar

The NotDecl abbreviation indicates the use of a symbol that has not been declared. Thus, the appearance of NotDecl in the type field indicates an error.

The Enable, Forward, ForwRout, Map, or Undeclar abbreviation refers to symbols that are declared elsewhere as routine or data-segment names. Thus, the appearance of any of these abbreviations in the type field usually indicates an error.

The Unbound abbreviation indicates that the compiler made no attempt to find a declaration for the symbol name because the name is not bound to a symbol. For example, a symbol in a macro actual-list, or in the false branch of a %IF compile-time conditional-function, is declared as Unbound.

As an example of using the symbol name and type fields, consider the following code segment:

```
00050    BEGIN
00051    LOCAL
00052         ALPHA,
00053         GAMMA;
           .
           .
           .
00080    END;
00081
00082    BEGIN
00083    LOCAL
00084        ALPHA;
           .
           .
           .
00095    END;
```

The appearance of the symbols in the cross-referencing listing would be as follows:

```
ALPHA    Local    52   .  .  .
         Local    84   .  .  .
GAMMA    Local    53   .  .  .
```

Defined Field

The defined field identifies the compiler listing line number of the declaration or the library (Lib) file number and has a fixed length of five characters. Exceptions occur with the NotDecl and Unbound symbol types. Because the symbol name in these cases cannot be associated with a declaration, no line number can appear, and the field is blank. The following example depicts the appearance of line numbers and a library file number in the defined field:

```
A          Own         5     20=
B          Own         6     20=
COMB       Macro   Lib01     20=
```

Note that a cross-reference map appears at the bottom of the listing which locates and identifies each compiled file (source, require, or library) by its initial line number and its system location (device, directory, and file specification).

Referenced Field

The referenced field lists additional references and uses of the symbol. Each entry consists of a 5-character line number (or a library file number) and a 2-character usage field. If the references require more than one line, the additional entries appear on subsequent lines.

The 2-character usage-fields describe the way in which the symbols are used. A usage-field may consist of none, one, or two of the following characters:

Flag	Meaning
Declaration-Usage	
e	EXTERNAL, EXTERNAL ROUTINE, or EXTERNAL LITERAL declaration
f	FORWARD or FORWARD ROUTINE declaration
m	MAP declaration
h	Condition handler enabling
u	UNDECLARE declaration
Data-Usage	
.	Fetch
=	Store
c	Routine call
a	Address use
@	Indirect use

A blank usage field indicates that usage is implied by the type of symbol, for example, a macro name used within a macro expansion or a structure name used as a structure-attribute in a declaration.

An (e), (f), (m), (h), or (u) flag appearing in the usage field indicates a reference to the symbol name within an EXTERNAL, FORWARD, MAP, ENABLE, or UNDECLARE type declaration.

The fetch flag (.) indicates that a data segment has been “fetched from” a location defined by the symbol name, while the store flag (=) indicates that a value has been “stored into” a location defined by the symbol name.

The address-use flag (a) indicates that the address of a data segment, defined by the symbol name, has been stored into another data segment. For example, A = B indicates that the address of the data segment defined by B is stored in data segment A. Thus, symbol B would be flagged (a) for its use as an address, and symbol A would be flagged (=) for its use as storage.

The indirect-use flag (@) never appears alone. This flag is always combined with the remaining data-usage flags to indicate that a data segment has been used indirectly, such as fetched from (@.) or stored into (@=); however, note that all multiple levels of indirection are flagged the same as a single level of indirection.

The following list provides samples of the 2-character data-usage codes. The examples reflect direct and indirect data uses of symbol B as a BLOCK structure and then as a REF BLOCK structure.

Code	B;BLOCK[n]	B:REF BLOCK[n]
A = .B	.	.
A = ..B	@.	@.
A = ...B	@.	@.
A = B[C]	a	@a
A = .B[C]	.	@.
B = .A	=	=
(.B) = .A	@=	@=
B[C] = .A	=	@=
B() = A	c	c
(.B) () = A	@c	@c

Thus, in relation to direct and indirect addressing, the utility recognizes ordinary structures and FIELD references. For example, consider the following code segment where explicit FIELD references are made to data segments:

```
00030    FIELD
00031       My_fields =
00032           SET
00033           This_field = [0,0,8,0],
00034           That_field = [0,1,8,0]
00035           TES;
00036    OWN
00037        B : REF BLOCK[] FIELD (My_fields);
00038
00039    B[This_field] = .B[That_field] + 1;
```

The cross-reference listings for B are as follows:

```
B              Own      37      39@=    39@.
 .
 .
 .
THAT_FIELD    Field     34      39.
THIS_FIELD    Field     33      39=
```

Note that because B is declared as a REF structure, the structure references to B are indirect references.

The next code example reflects an indirect address usage of B:

```
00030    FIELD
00031       My_fields =
00032           SET
00033           This_field = [0,0,8,0],
00034           That_field = [0,1,8,0]
00035           TES;
00036    OWN
00037        B : REF BLOCK[] FIELD (My_fields);
00038
  .
  .
  .
00108    C = B[That_field]
```

The listing for B is now as follows:

```
B           Own        37     108@a
 .
 .
 .
THAT_FIELD  Field      33     108
```

In this example, B points to a BLOCK in memory. Through B, an address within the BLOCK is indirectly stored in C; thus, an indirect address use is flagged for B.

2.3.3 Output Listing with Cross-Reference Listing

The listing in Example 2–7 includes a cross-reference listing that was produced by compiling module TEST with the following options:

```
$ BLISS/SOURCE_LIST:(REQUIRE)/LIST/CROSS_REFERENCE TEST
```

Note that the listing includes cross-referenced information for the LIBRARY and REQUIRE files. The reference list is followed by a cross-reference map, which specifies the first and last lines of the files used, and a flags legend, which describes the codes used in the referenced field.

Example 2–7 Output Listing with Cross-Reference Listing Included

```
TEST         2-Mar-1987 14:17:29    VAX BLISS--32 T4.3-793              Page   1
            16-Apr-1983 20:02:35    BLISS$:[ELLIS.B32]TEST.BLI;1            (1)

;   0001  0     module test (main = mainprog) =
;   0002  1     begin
;   0003  1
;   0004  1     own
;   0005  1         a,
;   0006  1         b;
;   0007  1     external routine
;   0008  1         error;
;   0009  1     library 'combn' ;
;   0010  1     require 'rfact' ;

TEST         2-Mar-1987 14:17:29    VAX BLISS--32 T4.3-793              Page   2
            13-Apr-1983 10:27:34    BLISS$:[ELLIS.B32]RFACT.REQ;2           (1)

;  R0011  1     ROUTINE FACT (N) =
;  R0012  1         IF .N GTR 1
;  R0013  1         THEN
;  R0014  1             .N * FACT(.N - 1)
;  R0015  1         ELSE
;  R0016  1            1;

                                               .TITLE  TEST

                                               .PSECT  $OWN$,NOEXE,2

                                 00000 A:     .BLKB   4
                                 00004 B:     .BLKB   4

                                              .EXTRN  ERROR

                                              .PSECT  $CODE$,NOWRT,2
```

(continued on next page)

Example 2–7 (Cont.) Output Listing with Cross-Reference Listing Included

```
                          0000 00000 FACT:   .WORD   Save nothing       ; 0011
            01       04   AC  D1 00002       CMPL    N, #1              ; 0012
                          0E  15 00006       BLEQ    1$                 ;
7E      04  AC            01  C3 00008       SUBL3   #1, N, -(SP)       ; 0014
        EF  AF            01  FB 0000D       CALLS   #1, FACT           ;
            50       04   AC  C4 00011       MULL2   N, R0              ;
                              04 00015       RET                        ;
            50            01  D0 00016 1$:   MOVL    #1, R0             ; 0012
                              04 00019       RET                        ; 0016

; Routine Size:  26 bytes,    Routine Base:  $CODE$ + 0000

TEST         2-Mar-1987 14:17:29    VAX BLISS--32 T4.3-793              Page   3
            16-Apr-1983 20:02:35    BLISS$:[ELLIS.B32]TEST.BLI;1               (1)

;   0017  1
;   0018  1     routine mainprog =
;   0019  2         begin
;   0020  2         a = combinations (3, 2);
;   0021  2         b = combinations (6, 4);
;   0022  2
;   0023  2         1                           ! VMS wants a success return
;   0024  1         end;


                             001C 00000 MAINPROG:
                                                   .WORD   Save R2,R3,R4   ; 0018
               54       E1   AF  9E 00002          MOVAB   FACT, R4        ;
                             03  DD 00006          PUSHL   #3              ; 0020
               64            01  FB 00008          CALLS   #1, FACT        ;
               53            50  D0 0000B          MOVL    R0, R3          ;
                             01  DD 0000E          PUSHL   #1              ;
               64            01  FB 00010          CALLS   #1, FACT        ;
               52            50  D0 00013          MOVL    R0, R2          ;
                             02  DD 00016          PUSHL   #2              ;
               64            01  FB 00018          CALLS   #1, FACT        ;
               50            52  C4 0001B          MULL2   R2, R0          ;
0000'   CF     53            50  C7 0001E          DIVL3   R0, R3, A       ;
                             06  DD 00024          PUSHL   #6              ; 0021
               64            01  FB 00026          CALLS   #1, FACT        ;
               53            50  D0 00029          MOVL    R0, R3          ;
                             02  DD 0002C          PUSHL   #2              ;
               64            01  FB 0002E          CALLS   #1, FACT        ;
               52            50  D0 00031          MOVL    R0, R2          ;
                             04  DD 00034          PUSHL   #4              ;
               64            01  FB 00036          CALLS   #1, FACT        ;
               50            52  C4 00039          MULL2   R2, R0          ;
0000'   CF     53            50  C7 0003C          DIVL3   R0, R3, B       ;
               50            01  D0 00042          MOVL    #1, R0          ; 0024
                                 04 00045          RET                     ;

; Routine Size:  70 bytes,    Routine Base:  $CODE$ + 001A

;   0025  1
;   0026  1     end
;   0027  0     eludom
```

(continued on next page)

Example 2–7 (Cont.) Output Listing with Cross-Reference Listing Included

```
;                                      PSECT SUMMARY
;
;  Name        Bytes                        Attributes
;
;  $OWN$       8  NOVEC,  WRT,  RD ,NOEXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)
;  $CODE$     96  NOVEC,NOWRT,  RD ,  EXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)

TEST         2-Mar-1987 14:17:29    VAX BLISS--32 T4.3-793              Page   4
            16-Apr-1983 20:02:35    BLISS$:[ELLIS.B32]TEST.BLI;1            (1)



Symbol         Type  Defined  Referenced ...
------------ -------- ----- ----------------
A            Own          5    20=
B            Own          6    21=
COMB         Macro    Lib01    20      21
COMBINATIONS Macro    Lib01    20      21
ERROR        ExtRout      8    20      21
FACT         Routine     11    14c     20c     21c
MAINPROG     Routine     18
N            RoutForm    11    12.     14.


                CROSS REFERENCE MAP

Line #      Event        File ...
-----  ---------------  ---------
    1  Source (start)   BLISS$:[ELLIS.B32]TEST.BLI;1
    1  Module TEST
    9  Library #1       BLISS$:[ELLIS.B32]COMBN.L32;1
   11  Require (start)  BLISS$:[ELLIS.B32]RFACT.REQ;2
   16  Require (end)
   27  Eludom TEST

        KEY TO REFERENCE TYPE FLAGS

            .   Fetch
            =   Store
            c   Routine call
            a   Address use
            @   Indirect use
            e   External, external routine, or external literal declaration
            f   Forward or forward routine declaration
            h   Condition handler enabling
            m   Map declaration
            u   Undeclare declaration

;                               Library Statistics
;
;                              -------- Symbols --------   Pages      Processing
;       File                   Total    Loaded   Percent   Mapped     Time
;
; BLISS$:[ELLIS.B32]COMBN.L32;1  2        2       100      6          00:00.1
```

(continued on next page)

Example 2–7 (Cont.) Output Listing with Cross-Reference Listing Included

```
TEST         2-Mar-1987 14:17:29    VAX BLISS--32 T4.3-793              Page   5
            16-Apr-1983 20:02:35    BLISS$:[ELLIS.B32]TEST.BLI;1            (1)

;                                      COMMAND QUALIFIERS

;       BLISS/SOURCE_LIST:(REQUIRE)/LIST/CROSS_REFERENCE TEST

; Size:         96 code + 8 data bytes
; Run Time:        00:01.7
; Elapsed Time:    00:20.3
; Lines/CPU Min:     947
; Lexemes/CPU-Min:  7298
; Memory Used:  28 pages
; Compilation Complete
```

2.4 Compilation Summary

The compilation summary appears at the end of every compilation listing and consists of the following information:

- The routine size and program section relative starting address (following each routine)
- A program section summary (at the end of the module)
- If a cross-reference listing is added, a cross-reference map of the files used and the line number where each file is first referenced
- If a cross-reference listing is added, a key to the meaning of the usage-field characters
- Library usage statistics indicating the libraries used and the number of names loaded from each library (omitted if no libraries are used)
- The number of memory pages mapped and the processing time
- The command line used to compile the module
- The number of warnings and errors (omitted if no warnings or errors exist)
- A summary of statistics for the module, consisting of size of code and data (in bytes); run time; elapsed time; number of lines and lexemes processed per CPU minute; memory used; and a statement that the compilation is complete

2.5 Error Messages

The BLISS compiler detects two types of errors: fatal and warning. A fatal error is one that the compiler cannot handle without potentially skipping some source. A warning error is one for which the compiler has an effective recovery technique that permits it to generate an executable object module. The warning and fatal error messages are listed separately in Appendix E. The warnings are listed by number, and each warning includes an explanation of the error and a recommended user action.

If a fatal error is detected, the compiler continues to check the syntax of the remainder of the program; any subsequent errors can be detected, but neither an object module nor the object part of the output listing is produced following the detection of the fatal error.

A warning error message begins with the indentification WARN. For example, the routine declaration for IFACT includes a coding mistake, as follows:

```
RESULT = .REULT*.I;
```

The BLISS compiler detects this error and reports the warning message shown in the following segment from the output listing:

```
;   0014  2              RESULT = .REULT*.I;
; WARN#000      ..........1  L1:0014
; Undeclared name:  REULT
```

The message is not fatal, because the compiler can declare the undeclared name REULT as EXTERNAL and continue processing without omitting the compilation of any source.

Consider a different kind of coding error, as follows:

```
ROUTINE RFACT (N =
```

The BLISS compiler detects this error and reports the message given in the following segment from the output listing:

```
;   0013  2         INCR I FROM 2 TO .NDO
; WARN#000      .....................1   L1:0013
; Undeclared name:  NDO
;   0014  2              RESULT = .REULT*.I;
; WARN#066      .........1  L1:0014
; Two consecutive operands with no intervening operator.
; A "DO" has been inserted
; WARN#000      .................1  L1:0014
; Undeclared name:  REULT
;   0015  2         .RESULT
;   0016  1         end
;   0017  1
;   0018  1     ROUTINE RFACT (n =
; ERR #071      2......3.........1  L1:0018  L3:0016
; Missing comma or closing bracket in formal parameter list for RFACT
; The incorrect delimiter was "="
```

Omitting the blank between the name N and the keyword DO caused another warning, while omitting the close parenthesis, that is (N =), caused one fatal error. With the absence of a blank seperator, the compiler sees NDO and RESULT as two consecutive operands with no intervening operator and inserts a DO. However, when the compiler fails to find the close parenthesis, it cannot make syntactic sense of the line; therefore, it reports a fatal error message and suppresses the production of an object file.

Note that, although the compiler continues to check the syntax of the remainder of the module, some text may be left unscanned. Also, the scan process can sometimes cause genuine errors to be missed and spurious errors to be reported. A module cannot be assumed to be fully checked by the compiler until all error messages are eliminated.

The BLISS compiler supplies a great deal of information in its error messages. Each error message occupies two lines; the first line classifies and pinpoints the error, and the second line gives a short description of the error. For example, consider the following error message from the above example:

```
;   0013  2         INCR I FROM 2 TO .NDO
; WARN#000      .....................1   L1:0013
; Undeclared name:  NDO
```

The first error message line indicates a warning (WARN) and provides the error message number 000, followed by a pointer to the place in the input line at which the error was detected and a line indicator. The second line describes the error.

The first line of an error message lines up with the input column at which the compiler detected the error. Under the preface for the input line, the error message has a preface part that gives the type of error (warning or fatal) and the error number. (Refer to Appendix E.) Under the text part of the input line, the error message can have up to three pointers and three line indicators. The pointers are numbered from 1 to 3 and the meaning associated with each of the pointers is given in the following list:

Pointer	Meaning
1	Indicates the point in the input text at which the error was detected
2	Indicates the beginning of the current control scope
3	Indicates the end of the last control scope that was successfully closed prior to the detection of the error

The line indicators are closely related to the pointers in meaning, but whereas the pointers indicate a position within a line, the line indicators indicate a line within the program, as follows:

Line Indicator	Meaning
L1:nnnn	Indicates the line nnnn in the input at which the error was detected
L2:nnnn	Indicates the line nnnn at which the current control scope begins
L3:nnnn	Indicates the line nnnn at which the last control scope was successfully closed

Line indicators are usually not too informative when the error is confined within a program line, as in the examples given above, but they are very useful for errors that span several lines. For example, consider the full source listing for the module TESTFACT given in Example 2–8. This version of TESTFACT includes the coding error illustrated in the above examples. The error message at the end of the program identifies with line indicators the point at which the error was detected (line 0032), the line at which the control scope began (line 0013), and the line at which the control scope was closed (line 0032).

With the information provided by the line indicators for error message 012, the source of the error is identified as the typing error in line 0013. For example:

Example 2–8 Example of Error Messages in Source Listing

```
    .
    .  (header)
    .
;   0001 0 MODULE TESTFACT (MAIN = MAINPROG)
;   0002 0 BEGIN
; WARN#048  1  L1:0002
; Syntax error in module head
;   0003  1
;   0004  1 OWN
;   0005  1     A,
;   0006  1     B;
;   0007  1
;   0008  1 ROUTINE IFACT (N) =
;   0009  2     BEGIN
;   0010  2     LOCAL
;   0011  2 RESULT;
;   0012  2     RESULT = 1;
;   0013  2     INCR I FROM 2 TO .NDO
; WARN#000  .....................1  L1:0013
; Undeclared name:  NDO
;   0014 2  RESULT = .REULT*.I;
; WARN#066  ........1  L1:0014
; Two consecutive operands with no intervening operator.  A "DO" has been inserted
; WARN#000  .................1  L1:0014
; Undeclared name:  REULT
;   0015  2     .RESULT
;   0016  1     END;
;   0017  1
;   0018  1 ROUTINE RFACT (N =
; ERR #071      2......3.........1  L1:0018  L2:0018  L3:0016
; Missing comma or closing bracket in formal parameter list for RFACT
; The incorrect delimiter was "="
;   0019  1     IF .N GTR 1
;   0020  1     THEN
;   0021  1 .N * RFACT(.N - 1)
;   0022  1     ELSE
;   0023  1 1;
;   0024  1
;   0025  1 ROUTINE MAINPROG =
;   0026  2     BEGIN
;   0027  2     A = IFACT(5);
;   0028  2     B = RFACT(5);
;   0029  2
;   0030  2     1 ! VMS wants a success return
;   0031  1     END;
;   0032  1 END
;   0033  0 ELUDOM

; Information:  0
; Warnings:     4
; Errors:       1
    .
    .
    .
```

3
Linking, Executing, and Debugging

This chapter describes the process of linking, executing, and debugging a BLISS program.

3.1 Linking

Before you can execute your program, you must use the VMS Linker to link the various pieces together to form an executable image. The linking process makes the connection between external variables referenced in one module and global variables defined in another module.

Some examples of linking are as follows:

- To link a single module, use the following command:

  ```
  $ LINK ALPHA
  ```

  In response to this command, the linker reads the object module in ALPHA.OBJ and creates the executable image ALPHA.EXE.

- To link the modules ALPHA, BETA, and GAMMA, use the following command:

  ```
  $  LINK ALPHA,BETA,GAMMA
  ```

  In response to this command, the linker combines the object module in the file ALPHA.OBJ with the object modules BETA.OBJ and GAMMA.OBJ to produce the single executable image ALPHA.EXE, which can then be executed.

The link operation is described in detail in the *VAX/VMS Linker Reference Manual*.

3.1.1 LINK Command Syntax

The syntax of the LINK command is as follows:

LINK [/qualifier...] object-file ,...

3.1.2 LINK Command Semantics

The linker reads the object modules contained in each file named in the link command to create a linked, executable image. The name of the executable image is taken from the name of the first object-file, and the file type EXE is used. If no file type is included in the file-spec, file type OBJ is assumed.

If the /DEBUG qualifier is used in the link-command, the VMS Debugger is linked with the specified object-files.

The linker has many qualifiers, but only the /DEBUG qualifier is discussed here. Other qualifiers are described in the *VAX/VMS Linker Reference Manual*.

Appearance of the following error message during the link operation:

```
%LINKER-W-TRUNC, Trunc. error in module <name>, P-section
<name>, offset %X <address>
```

generally implies that the program is larger than 32KB.

Specifying the following module qualifier causes the compiler to generate long displacement values for instruction operands:

```
ADDRESSING_MODE(EXTERNAL=LONG_RELATIVE, NONEXTERNAL=LONG_RELATIVE)
```

3.2 Executing a Linked Program

To run your program, use the executable image produced as a result of the link operation, as follows:

```
$  RUN ALPHA
```

Your program, ALPHA, then executes. Any input or output in your program takes place, and control then returns to DCL, which then prompts for another command.

3.3 Debugging

The VMS Debugger is used to debug or test BLISS programs. The following discussion of BLISS debugging assumes that you are familiar with using the debugger for VAX MACRO programs, as described in the *VAX/VMS Debugger Reference Manual*. This section describes BLISS debugging facilities primarily in terms of how such usage differs from that of the MACRO-level debugger.

The debugger provides a number of features specifically designed for more convenient debugging of BLISS programs. These BLISS–specific features are as follows:

- Debug expression syntax that is consistent with BLISS language syntax: radix control operators, arithmetic- and address-expression operators, field selectors, and so on
- Support of the fetch operator in arithmetic expressions
- Expression evaluation according to BLISS rules for operator precedence and use of parentheses
- BLISS–style references to elements and components of standard (predeclared) data structures: VECTOR, BLOCK, BLOCKVECTOR, and BITVECTOR; and support of REF data segments
- Support of field-names in structure references
- Support of BLISS field-references, as in EXAMINE X<15,3,0>

In combination, these features provide a high-level, BLISS–like facility for examining and modifying program data segments, both scalar and structured.

Many of the procedural aspects of BLISS debugging, however, such as setting breakpoints and program stepping, must necessarily be handled at object-code level; that is, with reference to the assembly listing, as in MACRO-level debugging. This is a consequence of two related characteristics of BLISS: It is not a statement language (as is FORTRAN, for example); and symbolic names for procedure addresses do not exist in BLISS below the routine level. In general,

the same procedure-related debugging facilities are provided for BLISS usage as for MACRO usage, and these facilities are exercised in very much the same way.

3.3.1 Initialized Modes and Types

When you initiate a debugging session for a BLISS program, the debugger provides the information message:

```
%DEBUG-I-INITIAL, language is BLISS, module set to 'ABC'
```

ABC represents your main module name. For BLISS, the initial entry/display modes are set to the following:

```
SYMBOLIC, HEXADECIMAL
```

You can change these mode settings with the SET MODE command and can restore them to the initial settings with the CANCEL MODE or SET LANGUAGE BLISS commands. The initialized modes are the same as those for MACRO.

The initial setting for the default type is as follows (refer to the SET TYPE command in Section 3.3.11.):

```
LONG INTEGER
```

The initial scope is PC scope.

3.3.2 Debugger Commands and Expression Syntax

Debugger commands are used for BLISS debugging exactly as they are for MACRO programs, except for the syntax within expressions. The debugger expression syntax is extended to allow use of a wide range of BLISS–style expressions, including most BLISS arithmetic and Boolean operators and the fetch operator, ordinary-structure-references (for predeclared structures), and BLISS field-references for specifying bit fields. (Debugger commands are summarized at the end of this chapter.)

Experienced VMS Debugger users should note in particular that, in expressions, the BLISS dot symbol (`.`) is recognized instead of the at sign (`@`) character as the debugger "contents" or indirect operator, and that the previous location symbol (`^`) is not recognized in BLISS mode.

The next two subsections describe special characters and keywords recognized by the debugger as operators and address symbols (or "address representation characters") in BLISS mode debugging. In some cases the characters are different from those recognized in MACRO mode with the same meaning, and in other cases the characters or keywords are an addition to those recognized in MACRO mode. These differences are noted. In all cases the operator usage is BLISS–like in that the meanings assigned to special characters and symbols in debugger commands are consistent with BLISS operator semantics.

Note that the significance of special characters as delimiters in debugger commands is the same for both BLISS and MACRO debugging. Also in order to maintain a reasonable correspondence with the *VAX/VMS Debugger Reference Manual*, the informal categories "arithmetic expression" and "address expression" are used below rather than the more formal BLISS language-syntax categories.

3.3.3 Operators in Arithmetic Expressions

Table 3–1 lists special characters and keyword symbols used in arithmetic expressions. The semantics, priority, and relationship of the operators listed in the table are as defined in BLISS. That is, the debugger evaluates expressions according to the language context. In particular, the intermediate and final results of an expression evaluation are calculated as longword (32-bit) values.

The following restrictions, relative to full BLISS expression syntax, apply to expressions in debug commands:

- Routine or function calls are invalid as part of an expression.
- The assignment operator (=) is invalid within an expression. (The equal sign character can be used only as a separator in DEPOSIT and DEFINE commands.)
- The BLISS relational operators (for example, EQL, NEQ, and LSS) are not supported.
- BLISS executable functions are not supported.
- Control expressions are not supported.
- Declarations are not supported.

Table 3–1 Arithmetic Expression Operators

Character	Interpretation
.	Fetch (prefix) operator
+	Arithmetic addition (infix) operator or (prefix) plus sign
–	Arithmetic subtraction (infix) operator or (prefix) minus sign
*	Arithmetic multiplication operator
/	Arithmetic division operator
^	Arithmetic shift operator (unlike MACRO syntax)
MOD	Arithmetic modulus operator
NOT	Boolean negation operator
AND	Boolean AND operator
OR	Boolean inclusive-OR operator
XOR	Boolean exclusive-OR operator
EQV	Boolean equivalence operator
. . .	Precedence operators; do (enclosed) first (unlike MACRO syntax)
%DECIMAL	Decimal radix operator (either %DEC or %DECIMAL)
%O 'string'	Octal radix operator
%X 'string'	Hexadecimal radix operator
%E 'string'	Single-precision floating-point operator

3.3.4 Special Characters in Address Expressions

Table 3–2 lists special characters that can be used to represent locations in a debugger expression. (Later subsections describe BLISS–style syntax extensions for structure references and field references.)

Table 3–2 Address Representation Characters

Character	Interpretation
.	When used alone or immediately before a delimiter, represents the last location addressed by an EXAMINE, DEPOSIT, SET BREAK, SET TRACE, or SET WATCH command. This is called the current location.
\	Represents the last value displayed by EXAMINE or EVALUATE. (The backslash is also used in forming path names, as in MACRO syntax.)
.	"Contents of" or indirect operator, when used as a prefix operator (unlike MACRO syntax).
:	Range operator (low address:high address) for the EXAMINE command.

3.3.4.1 Current Location Symbol (.)

Used either by itself or as an operand in an expression, the current location symbol (.) represents the location most recently referred to in an EXAMINE, DEPOSIT, SET BREAK, SET TRACE, or SET WATCH command. The value represented by the current location symbol remains unchanged until one of the above mentioned commands is used to refer to another location.

The symbol is assigned different meanings depending on the context in which it occurs. In addition to its meaning as current-location symbol, it is the debugger "contents" or indirect operator (as described below) and, analogously, the BLISS fetch operator in expressions. Thus, the sequence ".." itself constitutes a valid expression, meaning "contents of the current location". As an example, suppose that, at a given step in the execution of a program, the data-segment PTR contains the address of a longword vector named INBUF, and that the location last referred to by a debugger command is PTR. At this point the following command will display the address of INBUF (that is, the contents of location PTR):

```
EXAMINE .        / Examine current location /
```

The following command will display the contents of location INBUF:

```
EXAMINE ..       / Examine current location indirect /
```

Subsequently (since the last command altered the current-location value to location INBUF), the following command will display the contents of location INBUF+4:

```
EXAMINE .+4      / Examine current location plus 4 /
```

In contrast, note that the expression .(+4) would be interpreted (as in BLISS) as "contents of location 00004" rather than as "current location value plus four bytes".

The previous location symbol (^) recognized in MACRO debugging, is invalid when 'language is BLISS'. The address expression .-1 or .-4 can be used, for example, to represent the location of the "previous" byte or longword, respectively.

3.3.4.2 Last Value Displayed Symbol (\ )

You can use the backslash character (\ ) to represent the value most recently displayed. The value remains unchanged until the debugger displays a new value.

For example (again assuming the data segments PTR and INBUF previously described):

```
DBG>EXAMINE/DEC PTR
LISTER\LSTR\PTR: 1024)
DBG>EXAMINE/ASCII \
LISTER\INBUF<0,32>: ABCD
```

The first EXAMINE command displays the contents of location PTR as the value 1024. The display shows the scope of the symbol PTR to be module LISTER and routine LSTR. The second EXAMINE command displays the contents of location 1024, that is, the last value displayed. The display shows the location symbolized as INBUF with scope LISTER, and shows the contents of INBUF<0,32> to be "ABCD", interpreted as an ASCII string.

3.3.4.3 Contents Operator (.)

The dot character (.) used as an operator is the debugger "contents of" or indirect operator. It requests that the debugger evaluate the expression following it and then use the contents of the location addressed by the expression value rather than use the expression value itself. (The contents operator in MACRO debugging mode is the at sign (@).)

Examples of the use of the dot as both the current location symbol and the contents operator, depending on its context, are given in the two preceding subsections. A further example, based on the same assumptions, is as follows:

```
DBG>EXAMINE/ASCII .PTR
LISTER\INBUF: ABCD
```

Observe that this one EXAMINE command is equivalent to the sequence EXAMINE PTR followed by either EXAMINE \ or EXAMINE .. (in terms of the final location displayed).

Other examples, modified from those given in the *VAX/VMS Debugger Reference Manual*, follow:

```
DBG>DEPOSIT MASK = .MASK^4
```

This command shifts the current contents of the location MASK four bit positions to the left:

```
DBG>EXAMINE .R7 : .R7+%DEC'20'
```

This command displays the current contents of the 21 bytes beginning with the location addressed by general register 7.

3.3.4.4 Range Operator (:)

A colon (:) is used to separate elements in an address range in an EXAMINE command (see the previous example), just as in MACRO debugging.

Unlike its MACRO usage, however, the colon is not used in the EVALUATE command to express a bit-field specification. (In BLISS use, the standard BLISS field-selector notation is used instead, as described in Section 3.3.5.)

Several examples of range-operator usage, modified from those given in the *VAX/VMS Debugger Reference Manual*, follow:

```
DBG>EXAMINE/ASCII:1 INBUF : INBUF+6
DBG>EXAMINE . : . + %DEC'200'
DBG>EXAMINE/INSTRUCTION .PC : .PC+10
```

In the first command example, the address expression INBUF is interpreted as a nonstructured data reference because no access actuals are specified, although INBUF is assumed to name a structured (vector) data segment for the purposes of this discussion. This command simply displays the contents of the sequence of locations requested in storage units determined by the current length type, without reference to declared structure. In this case, since the specified type is ASCII:1, the first seven bytes beginning at location INBUF are displayed as follows:

```
LISTER\INBUF<0,8>: A
LISTER\INBUF+1<0,8>: B
        .
        .
        .
LISTER\INBUF+6<0,8>: G
```

If the type were ASCII:4, the first two longwords beginning at INBUF would be displayed, as follows:

```
LISTER\INBUF<0,32>: ABCD
LISTER\INBUF+4<0,32>: EFGH
```

The use of structure references in EXAMINE commands and the format of structured displays are described in Section 3.3.6.

3.3.4.5 Default Next Location Value

The default next location value is the address implied by the "shorthand" form of the EXAMINE command, that is, the EXAMINE verb (and possibly a type specification) immediately followed by a carriage return. The next location value is always the address of the byte following the last byte of the previous display item. For example, suppose the following EXAMINE commands are given, with the second command in "shorthand" form:

```
DBG>EXAMINE/ASCII:1 INBUF : INBUF+6
DBG>EXAMINE/ASCII:1
```

The following display would be produced:

```
LISTER\INBUF+7: H
```

If type ASCII:2 had been specified, the display would be as follows:

```
LISTER\>INBUF+8: IJ
```

If type ASCII:4 had been specified, the display would be as follows:

```
 LISTER\INBUF+0A: KLMN
```

3.3.5 Field References

A BLISS field reference (that is, an expression followed by a field selector) is used to specify a field of storage in EXAMINE, EVALUATE, and DEPOSIT commands. As in BLISS, the form of a field reference is as follows:

```
address<position,size,ext>
```

where ext is 0 for unsigned extension and 1 for signed extension; or

```
address<position,size>
```

where unsigned extension is assumed by default. The position and size values are interpreted as decimal integers, regardless of current radix mode, unless a radix operator is used (for example, %X'F'). The size value must not exceed 32 in any case.

In the EXAMINE command, a field selector has the same meaning as in the BLISS fetch context, because indirection (that is, contents of) is implied. That is, the EXAMINE command displays the contents of the field of storage specified by position and size, relative to the address given. (The field value is first extended to a longword, with signed or unsigned extension as requested). There is no syntactic restriction on the range of the position value, except that it must be a positive integer. For example:

```
DBG>EXAMINE/ASCII INBUF+8
LISTER\INBUF+8<0,32>: IJKL
DBG>EXAMINE/ASCII:1 INBUF<72,8>
LISTER\INBUF+9<0,8>: J
```

Note that the address expression INBUF+8 is equivalent to the field reference INBUF<64,32>.

In an EVALUATE command, the meaning of a field selector depends on whether a contents operator is specified (that is, whether indirection is requested). If the contents operator is specified, a field selector has the same meaning and lack of restriction on position as in the BLISS fetch context. For example:

```
DBG>EVALUATE/DEC .INBUF<72,8>
74
```

Observe that the effect of this field reference (with the contents operator specified) is the same as that produced by an equivalent EXAMINE command: the decimal value 74 is the ASCII code for the character "J".

If indirection is not specified in an EVALUATE command, a filed selector has the same meaning and restrictions as in the BLISS nonfetch, nonassignment content. That is, the expression *addr (p,s)* calculates the value addr+(p/8), where *p* must be a multiple of 8 and *s* is ignored. The effect of a field selector in this case is simply that of an address offset, with the position value *p* interpreted as p/8. For example, assume (as above) that the symbol INBUF has the value 1024:

```
DBG>EVALUATE/DEC INBUF<80,8>
1024
```

Note that the field reference INBUF<80,8> in the "nonfetch" context is equivalent to the expression INBUF+10 (as also is INBUF<80,16>, INBUF<80,23>, or INBUF<80,32>, for example).

In the DEPOSIT command, a field selector in the address expression on the left side of the equal sign has the same meaning and effect as in the BLISS assignment context. That is, it specifies a field of storage, relative to the given address, into which a value is to be stored. The stored value is truncated if necessary to fit the receiving field.

If a field selector appears in a "data" (righthand) operand of a DEPOSIT command, it has the same meaning and effect as described above for the EVALUATE command, depending upon whether a contents operator is specified.

3.3.6 Structure References

A BLISS–style structure reference can be used as a debugger address-expression if the data segment referred to has a standard BLISS predeclared structure (VECTOR, BITVECTOR, BLOCK, or BLOCKVECTOR). The debugger recognizes the following forms of structure reference:

- For VECTOR-structured segments:

```
segname[actual_1]
segname[field-name]
```

- For BITVECTOR-structured segments:

```
segname[actual_1]
segname[field-name]
```

- For BLOCK-structured segments:

```
segname[actual_1,actual_2,actual_3,actual_4]
segname[field-name]
```

- For BLOCKVECTOR-structured segments:

```
segname[actual_1,actual_2,---,actual_5]
segname[actual_1,field-name]
```

segname
Is a name declared with the appropriate structure-attribute or appropriate REF structure-attribute.

actual_i
Is an expression evaluated as a decimal integer representing a valid access-actual for the given data structure.

field-name
Is a name declared in a source-program FIELD declaration that represents a valid field-definition for the given data structure.

Note that access-actual values are interpreted in decimal unless a radix operator is used, regardless of the current radix mode.

The following EXAMINE commands provide a simple example of structure references and structured display format:

```
DBG>EXAMINE/ASCII INBUF[1]
LISTER\INBUF[1]: EFGH
DBG>EXAMINE/ASCII:16 INBUF[0]
LISTER\INBUF[0]: ABCDEFGHIJKLMNOP
DBG>EXAMINE/ASCII INBUF[0]:INBUF[3]
LISTER\INBUF[0]: ABCD
LISTER\INBUF[1]: EFGH
LISTER\INBUF[2]: IJKL
LISTER\INBUF[3]: MNOP
DBG>EXAMINE INBUF[1]:INBUF[2]
LISTER\INBUF[1]: 48474645
LISTER\INBUF[2]: 4C4B4A49
DBG>EXAMINE
LISTER\INBUF+0C<0,32>: 504F4E4D
```

In response to a structure reference, the debugger ignores default type and displays data in accordance with the allocation unit declared for the structure. For example, if you specify a range of data within a BLOCKVECTOR segment in an EXAMINE command (in terms of colon-separated structure references) and the segment is declared with allocation unit WORD, the debugger responds as follows:

1. Displays the content of the component specified by the first structure reference
2. Skips any remaining bits (up to seven) if the component does not end on a byte boundary
3. Displays any remaining data within the range in 2-byte units, beginning with the next location and ending with the last location occupied by any part of the component specified in the second structure reference (plus an additional byte, if required, to round up to a word-size display unit)

An appropriately defined field name can be used in place of explicit access-actual values, as shown in the structure-reference formats given above. No symbolic expression other than a field name is valid in a debugger structure reference.

In an EVALUATE command that specifies indirection, a structure reference results in an evaluation of the content of the requested element or component in current radix mode. The display differs as usual from EXAMINE in that the location of the evaluated item is not reported, and context type is ignored. For example:

```
DBG>EVALUATE .INBUF[2]
4C4B4A49
```

In an EVALUATE command without indirection (no contents operator specified), a structure reference results in an evaluation of the address corresponding to the requested element or component—that is, the address of the byte in which the data item begins. (In the case of a BLOCK or BLOCKVECTOR structure, the access actuals representing size and extension are ignored.) For a simple example, assume (as above) that the symbol INBUF names a longword vector segment at decimal location 1024:

```
DBG>EVALUATE INBUF
1024
DBG>EVALUATE INBUF[3]
1036
```

(This example assumes the current radix mode to be DECIMAL.)

For a somewhat more complex example, assume a block-structured segment named BLK2 begins at location 648 (decimal) and is allocated in longwords. Further, assume a set of declared field names F1, F2, F3, and F4. Also, for simplicity assume DECIMAL radix mode:

```
DBG>EVALUATE BLK2
648
DBG>EVALUATE F4
2, 4, 12, 0
DBG>EVALUATE BLK2[F4]
656
```

This example illustrates two facts:

- That the EVALUATE command evaluates a field name by reporting the access-actual values defined for that name. (Access-actual values, like field-selector parameters, are always displayed in decimal regardless of current radix mode.) The last value displayed (represented by a backslash) following this type of command is the last access-actual shown in the list: in this case, the value 0.
- That evaluating the structure-reference BLK2[2,4,12,0] results in the address value BLK2+8, because the component identified by the reference begins in the first byte of the third longword in segment BLK2. (More specifically, field F2 is defined as the 12 bits beginning with bit 4 of the third allocation unit.) As stated previously, the size and extension values are not involved in the evaluation.

In the DEPOSIT command, a structure reference used as the address expression on the left side of the equal sign has the same meaning and effect as in the BLISS assignment context. That is, it specifies an element or component of a data structure into which a value is to be stored. The stored value is truncated, if necessary, to fit the receiving element or component.

If a structure reference appears in a “data” (righthand) operand of a DEPOSIT command, it has the same meaning as described above for the EVALUATE command, depending on whether a contents operator is specified. That is, with indirection specified, the structure reference results in a fetch of the element or component indicated, with or without sign extension as requested. Without indirection, a structure reference results in the same kind of address evaluation as is performed by the EVALUATE command.

3.3.7 REF Structure References

The debugger recognizes and treats REF data segments in a manner consistent with the BLISS language. (A REF data segment is a longword segment declared with the attribute REF structure-name.) When a REF segment name is used in a structure reference, the debugger automatically supplies an extra level of indirection, treating the name as the location of a pointer to a segment that has the same structure as that declared for the REF segment.

As in BLISS, when a REF segment name is given in a debugger command without access actuals, the extra level of indirection is not provided. In this case, the name is interpreted as an ordinary address reference.

Note that the debugger recognizes and supports REF segments if they are declared with one of the predefined structure-names: that is, VECTOR, BITVECTOR, BLOCK, or BLOCKVECTOR. (User-defined structures are not supported as such by the debugger.)

3.3.8 Scope of Names

Whereas BLISS determines the scope of a data-segment name on a block basis, the debugger's smallest unit of scope is the routine. This causes no problems if a given name is declared only once within a routine. Ambiguities can arise, however, if a name declared at a given level (for example, at the outer routine level) is redeclared one or more times in contained blocks.

This includes the obscure case of redeclaring a routine formal name in a MAP declaration (in order to give it certain attributes, for example), because a formal name is effectively declared as a LOCAL scalar, with default attributes LONG and UNSIGNED, in the implicit block that surrounds every routine body. Redeclaration also includes the common case where a name, explicitly declared as a permanent data segment in a containing block, is redeclared (for example, LOCAL, STACKLOCAL, or REGISTER) in one or more contained blocks.

In all such cases, the debugger knows only of the last declaration of a given name that occurs in the source code for a routine; that is, the last declaration of the name encountered by the compiler when processing the source code for a particular routine. This means that the attributes of (and storage address corresponding to) a given name as known by the debugger are determined by that last declaration.

An entirely different and more subtle kind of problem can arise in connection with references to temporary data segments by name, a problem that is inherent in the compiler's optimization techniques. As described in Section 7.1.4, the compiler may allocate two or more temporary values to the same storage location or register at different points within a routine, if their "useful lifetimes" do not overlap. (The compiler does this to optimize the use of time and space.)

The possible implication of this for debugging is that, at a given point in the execution of a routine, a name declared as LOCAL, STACKLOCAL, or REGISTER can point to a location occupied by either a value corresponding to a different temporary data segment or a compiler-defined temporary value. The only way to resolve the resultant ambiguity (if examination of such a value is necessary) is by carefully interpreting the object code and examining the program counter when stepping through the routine in question.

3.3.9 Source-Line Debugging

When the BLISS–32 compiler is executed under VMS Version 4 (or higher versions), additional information is entered into the debugger symbol table, which permits a limited form of source line number debugging (where the source line numbers are defined as the numbers printed in the leftmost margin of a listing).

For example, it is possible to set a break on the first instruction of a source-line using a %LINE syntax:

```
DBG>SET BREAK %LINE 42
```

Note, however, that because BLISS–32 is an optimizing compiler for an expression language, you cannot access all source lines with the %LINE syntax.

However, using the TYPE command, you can display the source text for any line, including comments.

```
DBG>TYPE 17:19
Module TESTER
17: J= .J+1;
18: T= IFACT(5);  !Check out 5! = 120
19: RETURN .T+J
```

Note that the TYPE command shown displays source lines ranging from 17 through 19, using the module designated by the current scope setting.

It is also possible to examine the source code associated with a current PC location as follows:

```
DBG>EXAMINE/SOURCE .PC
18: T= IFACT(5);
```

3.3.10 Effect of Compilation and Link-Time Qualifiers

Using the qualifier /DEBUG in the link operation instructs the linker to replace the user program's starting address with the debugger's starting address. The executable image formed as a result causes the debugger to be mapped into the user program's address space. When it is executed, control first goes to the debugger instead of the user program.

The debugger is a shareable image. When a shareable image is linked into an executable image, it is unnecessary to copy the physical content of the shareable image. When a program that uses a shareable image is run, the copy from the shareable image file is also run. In reality, debugger modules are mapped into memory; they are not physically there. (Refer to the *VAX/VMS Linker Reference Manual.*)

The type of data you can access symbolically depends on the settings for the /TRACEBACK and /DEBUG qualifiers for the compilation. If the /NOTRACEBACK qualifier was given, no symbolic access is possible. If the /TRACEBACK and /NODEBUG qualifiers are given, only the names of globals, routines, modules, and program sections are available to the debugger. If the /TRACEBACK and /DEBUG qualifiers are given, you can examine BIND, GLOBAL, EXTERNAL, OWN, LOCAL, STACKLOCAL, and REGISTER data names in addition to the other names. Moreover, you can use field names in structure accesses, reference literal names in expressions, and listing source line-numbers to set breaks and examine BLISS source code.

Recall that the /TRACEBACK qualifier is a compilation default, while the /DEBUG qualifier is not.

3.3.11 Debugger Command Summary

This section summarizes commands that you can use to debug BLISS programs. The summary presents the commands in alphabetical order.

As in BLISS notation, braces ({}) enclose optional command elements; they are not part of the syntax. The optional repetition symbols " . . . " and ", . . . " also have the same meaning as in BLISS syntax definitions.

See SET MODE for entry/display mode keywords; see SET TYPE for data type keywords.

With the exception of ASCII character input, the debugger automatically converts lowercase input to uppercase.

“Address-expression” in the command syntax representations can be the path name (see SET SCOPE) of a symbol in your program, a numeric value, a symbol that you defined during this debugging session, a debugger special character, or an expression that combines any of these elements. “Address-expression” also includes BLISS–style field references and structure references.

The debugger supports command line continuation. A command line can contain up to approximately 500 characters, including nonprinting characters. You indicate continuation with the hyphen (-) as the last character before the carriage return. The debugger indicates a continued line by displaying an underline character as the first character on the line rather than the DBG> prompt.

CTRL/X refers to the simultaneous typing of the CTRL key and the respective character key, that is, C, Y, or Z (refer to the *VAX/VMS DCL Dictionary* for information on the complete list of CTRL functions). CTRL/X echoes at the terminal as ^X.

All commands preceded by an asterisk (*) are available only with VMS Version 3.

With the exception of CTRL functions, you terminate all command lines by pressing the RETURN key.

@filespec
In debug mode, denotes an indirect command file that causes the debugger to begin taking debugger commands from the indicated file. An indirect command file can be invoked wherever any other debugger command can be given. An indirect command file can contain any valid debugger command, including another indirect command. An EXIT command within an indirect command file cancels one level of indirection; an EXIT command given at terminal input level terminates the debugging process.

CALL name {(argument , . . .)}
Calls a routine by its symbolic name or by its virtual address (an address expression is invalid) with optional argument list. An argument list must be enclosed by parentheses.

CANCEL ALL
Cancel all breakpoints, tracepoints, watchpoints, and user-set entry/display modes. Type is set to its default value (long integer), and scope is set to its default value of zero. The initial entry/display modes are restored. This command does not change the current contents of the debugger’s symbol table (that is, those symbols acquired from program modules at debugger initialization or through use of the SET MODULE command, or any symbols that you defined during this debugging session). The current language is not changed.

CANCEL BREAK address-expression
CANCEL BREAK/ALL
Cancels breakpoint set at specified address, or cancels all breakpoints.

CANCEL EXCEPTION BREAK
Cancels the request that your program stop at an exception condition (such as at a breakpoint).

CANCEL MODE
Restores initial entry/display modes. This command does not change the scope or the current language.

CANCEL MODULE module-name-list
CANCEL MODULE/ALL
Purges symbolic information associated with the named modules from the symbol table. The typical use is to make space available for symbols associated with another module or modules (see SET MODULE). Global symbols and any symbols defined during this debugging session are not affected.

CANCEL SCOPE
Sets the scope to zero.

***CANCEL SOURCE{/module=modnam}**
Cancels the current source directory search list established by previous SET SOURCE commands. When the qualifier (/module=modname) is used, the command cancels the effect of any previous command in which the same module name was specified, but does not affect commands in which other module names are specified or commands where the qualifier is not used.

CANCEL TRACE address-expression
CANCEL TRACE/CALL
CANCEL TRACE/BRANCH
CANCEL TRACE/ALL
Cancels a tracepoint set at the specified address, cancels all opcode tracing at call-type instructions, cancels all opcode tracing at branch-type instructions, or cancels all tracepoints and opcode tracing.

CANCEL WATCH address-expression
CANCEL WATCH/ALL
Cancels a watchpoint set at the specified address, or cancels all watchpoints.

CTRL/C
Has same effect, and echoes at terminal, as CTRL/Y (see below) if your program does not include an exception condition handler for CTRL/C.

CTRL/Y
Interrupts the debugger or executing program and transfers control to DCL, as signaled by the system prompt ($). Type DEBUG after the system prompt to return control to the debugger. Type CONTINUE after the system prompt to return control to the interrupted program. Typing any DCL command other than DEBUG or CONTINUE will probably force the premature exit of your program. You can use CTRL/Y to interrupt a looping program. To determine the point at which you interrupted your program, type the following:

```
DBG>EXAMINE/INSTRUCTION .PC
```

CTRL/Z
Same result as EXIT; that is, terminates the debugging session and transfers control to DCL.

DEFINE symbol-name=value {,symbol-name=value , . . . }
Equates the names on the name=value list with associated values for use during the debugging session. The debugger searches these symbols first when it

requires either a definition for a symbolic entry or a symbolic name to report a location.

DEPOSIT{/modifier . . . } address-expression=data{,data , . . . }
Enters the data specified in the data list in the sequence of locations that begins with the specified address. The modifier can be any mode or type keyword. (Mode and type keywords can be mixed.)

EVALUATE{/mode . . . } expression , . . .
Transforms input (which can be a field name, arithmetic expression, ASCII string, VAX MACRO instruction, symbol, structure-reference, field-reference, or numeric value) to associated values and displays the results. This command can also be used for such things as a desk calculator, radix converter, or symbol verifier. The debugger displays results in the order in which you specify the input. The displayed decimal value of the field name is a comma list of field components.

EXAMINE{/modifier . . . } address{:address}{,address{:address} , . . . }
Displays the current contents of the specified addresses. The colon signifies the range; that is, it displays the contents of the addresses from the low adddress through the high address. The modifier can be any mode or type keyword. (Mode and type keywords can be mixed.)

EXIT
Terminates the debugging session and transfers control to DCL. An EXIT command within an indirect command file cancels one level of indirection; an EXIT command given at terminal input level terminates the debugging process.

GO {address-expression}
Starts or continues program execution. The first GO command (without an address) starts the program at its transfer address. GO commands thereafter continue execution from a stopped point (as at a breakpoint or watchpoint, or because of an exception condition).

An address entry replaces the current program counter (PC) contents; execution starts or continues from the new location.

Once you have started a program, you should not attempt to restart at the transfer address or any other address. Program behavior is unpredictable when restarted.

***HELP topic {subtopic . . . }**
Displays a description, format, qualifiers, and parameters of a debugger command as specified by the topic parameter; and displays qualifiers and/or parameters as specified by the subtopic parameter.

***SEARCH {/qualifier {/qualifier} } range string**
Limits the debugger's search (of a source file) to specified program regions for occurrences of the string. The qualifiers are as follows:

/ALL
Specifies a search for all occurrences of the string and a display of every line in the specified range.

/NEXT
Specifies a search and display of only the first occurrence of the string in the specified range. (This is the default.)

/IDENTIFIER
Specifies a search for an occurrence of a string in the range specified. The string is displayed only if it is delimited by a character that is not part of an identifier in the current language.

/STRING
Specifies a search and display of an occurrence of the specified string. (This is the default.)

The range parameter limits the search to a specified program region as follows:

modname	Search the specified module from line number zero to the end of the module.
modname\line-num	Search the specified module from the specified line number to the end of the module.
modname\line-num:line-num	Search the specified module from the first line number given to the last.
line-num	Search the module designated by the current scope setting from the specified line number to the end of the module.
line-num:line-num	Search the module designated by the current scope setting beginning at the first line number and ending at the last.
<nothing>	Search the same module from which a source line was recently displayed (by a TYPE or EXAMINE/SOURCE command), beginning at the first line following the line displayed and continuing to the end of the module.

The string parameter specifies the string in the source code for which the search is initiated.

SET BREAK address-expression {DO (command list)}
Establishes a breakpoint at specified address (the breakpoint stops your program before the instruction beginning with “address-expression” is executed).

The debugger executes commands in DO sequence command format whenever your program stops because of the specified breakpoint. Parentheses are required as command list delimiters. Multiple commands must be separated by semicolons. Any complete debugger command can be used in this context, including GO, STEP, or CALL. If GO, STEP, or CALL is specified, it must be the last command in the sequence.

You can specify the /AFTER qualifier to defer a breakpoint as follows:

```
SET BREAK/AFTER:decimal-integer address-express {DO - - -}
```

The breakpoint is ignored until the nth pass through the specified location, as in an iteration, where n is within the range 1 through 32767. Thereafter, the breakpoint takes effect each time the debugger encounters it. You can specify a temporary (or one-time) breakpoint with the following:

```
SET BREAK/AFTER:0 address-expression
```

The first time the breakpoint is encountered, the debugger stops your program and cancels the breakpoint.

SET EXCEPTION BREAK
Stops the program and reports the current program counter contents if an exception condition occurs that was not initiated by a debugger command.

SET LANGUAGE language-name
Lets the debugger interpret input and display output in the syntax defined for the specified language. The debugger rejects commands that are invalid in the specified syntax. The debugger initially recognizes the language of the first module in your program that contains symbol information.

SET LOG file-name
Specifies a name other than the default name (DEBUG.LOG) for the debugger log file. The command can also be used to generate multiple log files during a single debugging session.

***SET MARGIN rm**
***SET MARGIN lm:rm**
***SET MARGIN lm:**
***SET MARGIN :rm**
Specifies the leftmost and rightmost source line character positions at which to begin and end a line of source code. The default value for the left margin is 1, and the default value for the right margin is 255.

***SET MAX_SOURCE_FILES n**
Specifies the maximum number of source files that the debugger can keep open at any one time.

SET MODE mode-keyword{,mode-keyword , . . . }
Allows or inhibits the entry and display of data in specified formats.

The following list describes the function of each keyword:

DECIMAL	Interpret/display data in decimal radix.
HEXADECIMAL	Interpret/display data in hexadecimal radix.
NOSYMBOLIC	Inhibit display of symbolic addresses.
OCTAL	Interpret/display data in octal radix.
SYMBOLIC	Display symbolic addresses.

The debugger's initial modes are SYMBOLIC and HEXADECIMAL.

You can also enter the mode keywords with the commands DEPOSIT, EVALUATE, and EXAMINE to override the current associated mode (radix and symbolic/nosymbolic). A slash must precede each mode keyword entered after these command verbs:

```
command-verb/keyword/keyword ...
```

SET MODULE module-name-list
SET MODULE/ALL
Enters nonglobal symbols and program section names associated with the program-module list into the debugger's symbol table, or enters information from all modules into the symbol table. The debugger cannot interpret nonglobal symbols unless their associated module names appear in the status report produced by the SHOW MODULE command with a "yes" indication.

SET OUTPUT keyword{,keyword, . . . }
Turns the logging function on or off, displays or inhibits the display of debugger commands executed from indirect command files or breakpoint actions, and permits or suppresses debugger output, except error messages, to the screen. Valid keywords are as follows:

LOG	Copy all command input (verbatim) and debugger output, including error messages, to the current log file; copy verified lines if VERIFY is specified.
NOLOG	Inhibit the creation of a debugger log file.
TERMINAL	Print all debugger output on the screen, including command input, debugger responses, comment lines, and warning and error messages.
	Print only error messages; suppress all debugger output, including verified lines.
VERIFY	Echo command lines on the terminal as they are executed from a breakpoint action or indirect command file.
NOVERIFY	Do not echo command lines from indirect command file on the screen.

The initial keyword settings are NOLOG, NOVERIFY, and TERMINAL. If default settings are used, no log file is produced, the printing of any command taken from an indirect command file and breakpoint action is inhibited, and all DEBUG output is printed on the terminal.

Specifying NOLOG and NOTERMINAL causes a warning message to be printed; output is printed on the terminal.

The following symbols may appear in the log file and the screen:

!	Indicates a debugger response
!!	Indicates a comment line

To log all debugger command I/O in the log file, use the following command:

```
SET OUTPUT LOG
```

SET SCOPE module-name{\routine-name . . . }
Establishes an ordered list of path names for use with the debugger's symbol search rules. A path name completely and unambiguously identifies a symbol. For BLISS, a path name is one of the following:

```
symbol-name
module-name{\routine-name ...}\symbol-name
```

The debugger evaluates an expression in which a symbolic entry appears only if a definition was located for the entry. If it fails to locate a match for a path name, the debugger reports the search failure and the symbol name.

Special path names are available for use in the list of path names of a SET SCOPE command:

Path Name	Meaning
0	Indicates the path name of the lexical entity. For example, routine or block that contains the current program counter.
1, 2, 3, . . .	Path name 1 indicates the caller of the lexical entity containing the current PC; path name 2 indicates the caller of path name 1, and so on.
\	The path name \ preceding a symbolic name indicates a global symbol of that name. For example: `EXAM \GLOBAL`This causes a search for a global symbol whose name is GLOBAL; nonglobal symbols of the same name are ignored.

***SET SEARCH parameter{,parameter}**
Establishes search parameters whenever a SEARCH command qualifier is not specified. The parameters determine the search for occurrences of a string as

follows: find all occurrences (ALL); find only the next occurrence (NEXT); display all occurrences found (STRING); display only the occurrences unbounded by a current language identifier (IDENTIFIER).

***SET SOURCE{/module=modname} dirname{,dirname . . . }**
Directs the search of specified directories for the source files. The command is used to locate source files that have been removed from compile-time directories.

SET STEP keyword {,keyword , . . . }
Establishes default conditions for the STEP command. Valid keywords are as follows:

INSTRUCTION	Steps increment in VAX MACRO instructions.
INTO	Allows stepping through called routine.
LINE	Steps increment by listing line numbers.
OVER	Steps over called routine (make call transparent).
*SOURCE	Displays the line of source code that corresponds to the instructions being executed.
*NOSOURCE	Inhibits the display of the line of source code corresponding to the instruction being executed.
SYSTEM	Allows stepping in system space.
NOSYSTEM	Inhibits stepping in system space (make execution transparent).

The initialized conditions for BLISS are INSTRUCTION, NOSYSTEM, and OVER.

SET TRACE address-expression
SET TRACE/CALL
SET TRACE/BRANCH
Sets a tracepoint at the specified address, or specifies tracing of all call-type or branch-type instructions. At a tracepoint, the debugger reports the current program counter contents and then continues program execution automatically.

SET TYPE
Specifies a data type to be associated to data when the debugger cannot infer a type from its input.

The following list describes the function of each keyword:

ASCII:n	Interpret/display data as a string of n ASCII characters, where n is an integer.
BYTE	Interpret/display data in byte units.
INSTRUCTION	Interpret/display VAX MACRO instructions.
LONG	Interpret/display data in longword units.
WORD	Interpret/display data in word units.

The debugger's initial type setting is LONG INTEGER. You can also enter the type keywords with the commands DEPOSIT and EXAMINE to override the current associated type. A slash must precede each type entered after these command verbs.

```
command-verb/keyword/keyword ...
```

SET WATCH address-expression

Reports if the contents of the specified locations are modified. The locations watched can be individual addresses (including the number of bytes specified by the length type in effect when the watchpoint was set) or the number of bytes associated with the symbol's data type (for example, double precision: eight bytes).

When the contents of a watched location changes, the debugger stops the program (as at a breakpoint) and reports both the previous contents and the current contents of the location.

SHOW BREAK

Reports the locations of current breakpoints and any relevant information associated with them, such as DO command sequences and "after" options.

SHOW CALLS {n}

Reports current call level and the hierarchy of call levels that preceded it (that is, traces your program's call history). If n (a decimal integer) is expressed, the debugger reports n call levels back from the current level (n has the range 0 through 32767). If n is omitted, all preceding call levels are reported.

SHOW LOG

Displays the name and status of the log file (see SET LOG.) The display appears as follows:

```
[not] logging to 'filename'
```

*SHOW MARGIN

Displays the current source line margin settings that are being used for the display of source code. The margin settings are established by the SET MARGIN command. The default margin settings are left margin 1 and right margin 255.

*SHOW MAX_SOURCE_FILES

Displays the maximum number of source files that the debugger can keep open at one time.

SHOW MODE

Reports the current entry/display modes (see SET MODE).

SHOW MODULE

Lists program modules by name, indicates (by yes or no) whether their associated symbol data exists in the debugger's symbol table and indicates the approximate space required for the entry of each module's symbol data. Lists also the amount of space currently unused. The debugger has no knowledge of any program module not reported in this status report.

SHOW OUTPUT

Displays the current output attributes and the name and status of the current log file. (See SET OUTPUT and SET LOG.) This command generates the following status report:

```
output: [no]verify, [no]terminal, and [not] logging to 'filename'
```

*SHOW SEARCH

Displays the current search parameters established by the SET SEARCH command or the default values of ALL and STRING.

SHOW SCOPE
Reports the current contents of SCOPE.

***SHOW SOURCE**
Displays the current source directory search lists established by the SET SOURCE or SET SOURCE/module=modname command.

SHOW STEP
Reports current default conditions for STEP (see SET STEP).

SHOW TRACE
Reports the locations of current tracepoints or that opcode tracing is in effect.

SHOW TYPE
Reports the current type setting.

SHOW WATCH
Reports the locations of current watchpoints and the number of bytes monitored by each watchpoint.

STEP{/keyword} {decimal-integer}
Begins program execution and then stops after executing the specified number of instructions or listing lines. (If you do not specify a count, single stepping is the default.) The count is a decimal integer between 0 through 32767.

The following keywords can either be used after the STEP command verb (STEP/keyword) or be set with the SET STEP command to establish the default conditions for STEP. The SHOW STEP command displays the current defaults.

The keywords have the following relationships:

SYSTEM/NOSYSTEM	Count/do not count steps in system space.
INTO/OVER	Count/do not count steps within a called routine.
INSTRUCTION	Step by instructions.
LINE	Step by listing line numbers.
*SOURCE/NOSOURCE	Display/do not display lines of source code.

The initialized defaults for BLISS are INSTRUCTION, NOSYSTEM, and OVER.

***TYPE { {modname\}line-number{:line-number} -**
{,{modname\}line-number}:line-number} . . . } }
Displays the source language statements corresponding to the line numbers specified.

In effect, you can read all the source language statements in the program by specifying a starting line number of 1 and an ending line number that is equal to or greater than the largest line number in the program listing.

If a module name and either a single line number or a range of numbers (separated by a colon) are not specified, the default scope setting is used to determine which module to use. The default scope is either the module designated by a SET SCOPE command or the module containing the current PC.

4

Machine-Specific Functions

Machine-specific functions (also called built-in functions) allow you to perform specialized VAX operations within the BLISS language. A machine-specific function call is similar to a BLISS routine call. It requires parameters and, in some cases, returns a value. If a machine-specific function that does not return a value is used in a context that requires a value, an error is reported, as in a BLISS routine.

Compiling a machine-specific function generates inline code, often a single instruction, rather than a call to an external routine. The compiler attempts to optimize the code it produces for a machine-specific function call by choosing the most efficient instruction sequence. In some cases, the optimization procedure results in a machine instruction being generated that is different from the one specified in the call.

Machine-specific functions in BLISS–32 are divided into categories, as illustrated in Table 4–1. A separate description of each function is given in the following sections. For a detailed discussion, consult the *VAX Architecture Handbook*.

Input Parameter Usage

The definitions of these functions require addresses as input parameters, even where values could have been specified for what VAX terms “source operands”, which are a longword or less in size. For example, where a length is required as the second input-parameter of the PROBER function, the call is written as follows:

```
   PROBER(...,UPLIT(5),...)
or
   PROBER(...,%REF(.A+2),...)
```

(The %REF is preferred, as the compiler is free to create immediate or short-literal addressing modes.)

Most functions also allow the name of a register to be used as the parameter, even though register names do not have values in BLISS–32. For example:

```
REGISTER R;
...
IF PROBER(...,R,...) THEN ...
```

is equivalent to:

```
REGISTER R;
...
IF PROBER(...,%REF(.R),...) THEN ...
```

The following are general rules for the use of %REF and undotted register names with the input parameters:

- A %REF value cannot be used with a DESTINATION address.

- With the exception of their use with TESTBITxx built-in functions, an undotted register can only be used for operands that are BYTE, WORD, LONG, or single-precision floating-point.
- A register name cannot be used as the address of a character string or packed decimal string.

Output Parameter Usage

Output parameters allow the return of register contents that result from executing certain machine-specific functions (such as CALLG).

For example:

```
BUILTIN
CALLG;
BIND
ARGLIST= UPLIT(1, PI);
....
CALLG( ARGLIST, MTH$DSIN; SINX[0], SINX[1] );
```

This code expands to the following:

```
CALLG ARGLIST, MTH$DSIN
MOVQ R0, SINX
```

Notice in the syntax of the machine-specific function that output parameters are separated from input parameters by a semicolon.

Table 4–1 Machine-Specific Functions

Function	Description
Processor Register Operations	
MFPR	Move from processor register
MTPR	Move to processor register
Parameter Validation Operations	
PROBER	Probe read accessibility
PROBEW	Probe write accessibility
Program Status Operations	
BICPSW	Bit clear PSW
BISPSW	Bit set PSW
MOVPSL	Move from PSL
Queue Operations	
INSQUE	Insert entry into queue
INSQxI	Insert entry into queue interlocked
REMQUE	Remove entry from queue
REMQxI	Remove entry from queue interlocked

(continued on next page)

Table 4–1 (Cont.) Machine-Specific Functions

Function	Description
Bit Operations	
FFC	Find first clear bit
FFS	Find first set bit
TESTBITCC	Test for bit clear; clear bit
TESTBITCCI	Test for bit clear; clear bit interlocked
TESTBITCS	Test for bit clear; set bit
TESTBITSC	Test for bit set; clear bit
TESTBITSS	Test for bit set; set bit
TESTBITSSI	Test for bit set; set bit interlocked
Arithmetic Operations	
ADAWI	Add aligned word interlocked
ADDD	Add D-floating operands
ADDF	Add F-floating operands
ADDG	Add G-floating operands
ADDH	Add H-floating operands
ADDM	Add multiword operands
ASHQ	Arithmetic shift quad
DIVD	Divide D-floating operands
DIVF	Divide F-floating operands
DIVG	Divide G-floating operands
DIVH	Divide H-floating operands
EDIV	Extended-Precision divide
EMUL	Extended-Precision multiply
MULD	Multiply D-floating operands
MULF	Multiply F-floating operands
MULG	Multiply G-floating operands
MULH	Multiply H-floating operands
SUBD	Subtract D-floating operands
SUBF	Subtract F-floating operands
SUBG	Subtract G-floating operands
SUBH	Subtract H-floating operands
SUBM	Subtract multiword operands

(continued on next page)

Table 4–1 (Cont.) Machine-Specific Functions

Function	Description
Arithmetic Comparison Operations	
CMPD	Compare D-floating operands
CMPF	Compare F-floating operands
CMPG	Compare G-floating operands
CMPH	Compare H-floating operands
CMPM	Compare multiword operands
Arithmetic Conversion Operations	
CVTDF	Convert D-floating to F-floating
CVTDI	Convert D-floating to integer
CVTDL	Convert D-floating to long
CVTFD	Convert F-floating to D-floating
CVTFG	Convert F-floating to G-floating
CVTFH	Convert F-floating to H-floating
CVTFI	Convert F-floating to integer
CVTFL	Convert F-floating to long
CVTGF	Convert G-floating to F-floating
CVTGH	Convert G-floating to H-floating
CVTGL	Convert G-floating to long
CVTHF	Convert H-floating to F-floating
CVTHL	Convert H-floating to long
CVTID	Convert integer to D-floating
CVTIF	Convert integer to F-floating
CVTLD	Convert long to D-floating
CVTLF	Convert long to F-floating
CVTLH	Convert long to H-floating
CVTRDL	Convert rounded D-floating to long
CVTRFL	Convert rounded F-floating to long
CVTRGL	Convert rounded G-floating to long
CVTRHL	Convert rounded H-floating to long
Character String Operations	
CMPC3	Compare characters 3 operand
CMPC5	Compare characters 5 operand
CRC	Cyclic redundancy calculation
LOCC	Locate character
MATCHC	Match characters
MOVC3	Move character 3 operand

(continued on next page)

Table 4–1 (Cont.) Machine-Specific Functions

Function	Description
Character String Operations	
MOVC5	Move character 5 operand
MOVTC	Move translated characters
MOVTUC	Move translated until character
SCANC	Scan characters
SKPC	Skip character
SPANC	Span characters
Decimal String Operations	
ASHP	Arithmetic shift and round packed
CMPP	Compare packed
CVTLP	Convert long to packed
CVTPL	Convert packed to long
CVTPS	Convert packed to leading separate numeric
CVTPT	Convert packed to trailing numeric
CVTSP	Convert leading separate numeric to packed
CVTTP	Convert trailing numeric to packed
EDITPC	Edit packed to character
MOVP	Move packed
Miscellaneous Operations	
BPT	Breakpoint
BUGL	Bugcheck with long operand
BUGW	Bugcheck with word operand
CALLG	Call with general argument list
CHMx	Change mode
HALT	Halt processor
INDEX	Index (subscript) calculation
NOP	No operation
ROT	Rotate a value
XFC	Extended function call

The compiler attempts to optimize the code that corresponds to a machine-specific function call. It chooses instruction sequences based on the type of result required of the function and the context in which it is used. This is illustrated in the following examples.

```
BLISS Source:

   IF TESTBITSS( X<3,1> )
   THEN
     BEGIN   !
     ...     ! CONSEQUENCE
     END     !
```

```
Generated Code:

   BBCS  #3,X,1$
   ...       ;
   ...       ; CONSEQUENCE
   ...       ;
1$:
```

Note that the compiler has chosen an instruction with the opposite sense to the machine-specific function. If 1$ could not be reached by a byte displacement, the expansion would be as follows:

```
   BBSS   3,X,1$
   BRW   2$
1$:

   ...      ;
   ...      ; CONSEQUENCE
   ...      ;
2$:
```

In the above example, the TESTBITSS routine is used in a situation where it is required to generate only a control flow result. If a real result actually was needed, it might be generated as in the following example.

```
 BLISS Source:

    Y = TESTBITSS( X<3,1> )

Generated Code:

    CLRL   Y
    BBCS   #3,X,1$
    INCL   Y
1$:
```

A machine-specific function differs from a routine call in that the compiler can determine exactly how the parameters of the routine call are going to be used, and so there are cases where %REF does not need to allocate a temporary storage. For example:

```
   BISPSW( %REF (%X'80'));   ! ENABLE DECIMAL OVERFLOW
```

This call would be coded without any temporary being allocated, as follows:

```
   BISPSW #^X80         ; ENABLE DECIMAL OVERFLOW
```

The compiler can make similar decisions when you use the undotted name of a local as a parameter in a machine-specific function call. You are syntactically specifying an address, but depending on the parameter, that local might still be allocated in a register. For example:

```
   MOVPSL(ALOCAL)
```

This might result in the following code:

```
   MOVPSL R2
```

4.1 ADAWI—Add Aligned Word Interlocked

ADAWI (SRCADDR, DSTADDR)

Parameters:

SRCADDR
Address of a word whose contents are added to the destination.

DSTADDR
Address of a word to which the source is to be added. The address must be word aligned (that is, the low bit must be zero).

Result:

Contents of the PSL

4.2 ADDD—Add D-Floating Operands

ADDD (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A
Address of a double-precision floating-point quadword used as the addend.

SRC2A
Address of a double-precision floating-point quadword used as the augend.

DSTA
Address of a quadword where the sum is stored.

Result:

NOVALUE

4.3 ADDF—Add F-Floating Operands

ADDF (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A
Address of a single-precision floating-point longword used as the addend.

SRC2A
Address of a single-precision floating-point longword used as the augend.

DSTA
Address of a longword where the sum is stored.

Result:

NOVALUE

4.4 ADDG—Add G-Floating Operands

ADDG (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A
Address of an extended double-precision floating-point quadword used as the addend.

SRC2A
Address of an extended double-precision floating-point quadword used as the augend.

DSTA
Address of a quadword where the sum is stored.

Result:
NOVALUE

4.5 ADDH—Add H-Floating Operands

ADDH (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A
Address of an extended-exponent double-precision floating-point octaword used as the addend.

SRC2A
Address of an extended-exponent double-precision floating-point octaword used as the augend.

DSTA
Address of an octaword where the sum is stored.

Result:
NOVALUE

4.6 ADDM—Add Multiword Operands

ADDM (SIZE, SRC1A, SRC2A, DSTA)

Parameters:

SIZE
Compile-time constant expression indicating the size of the operands in longwords (BLISS value units).

SRC1A
Address of extended multiprecision integer used as the addend.

SRC2A
Address of extended multiprecision integer used as the augend.

DSTA
Address of the sum of source 1 and source 2.

Result:
NOVALUE

4.7 ASHP—Arithmetic Shift and Round Packed

ASHP (SCALE, SRCLEN, SRCADDR, ROUND, DSTLEN, DSTADDR [;OUT0, OUT1, OUT2, OUT3])

Input Parameters:

SCALE
Address of a byte whose contents specify the scale count by a power of 10 (a positive count effectively multiplies while a negative count divides, a zero count moves and affects condition codes).

SRCLEN
Address of a byte whose contents specify the length of the source string to be scaled.

SRCADDR
Address of a quadword containing the quantity to be scaled.

ROUND
Address of a byte whose contents specify the decimal round operand (only used when a negative count is specified).

DSTLEN
Address of a byte whose contents specify the length of the destination string where the result is to be stored.

DSTADDR
Address of a quadword where the scaled result is to be stored.

Output Parameters:

OUT0
Value returned in R0 is zero.

OUT1
Value returned in R1 is the address of the byte containing the most significant bit of the source string.

OUT2
Value returned in R2 is zero.

OUT3
Value returned in R3 is the address of the byte containing the most sugnificant digit of the destination string.

Result:

NOVALUE

4.8 ASHQ—Arithmetic Shift Quad

ASHQ (SHIFT, SRCADDR, DSTADDR)

Parameters:

SHIFT
Address of a byte whose contents specify the shift count.

SRCADDR
Address of a quadword containing the quantity to be shifted.

DSTADDR
Address of a quadword where the shifted result is to be stored.

Result:

Contents of the PSL

4.9 BICPSW—Bit Clear PSW

BICPSW (MASKADDR)

Parameters:

MASKADDR
Address of a word whose contents are to be ones complemented and ANDed into the processor status word (PSW).

Result:
NOVALUE

4.10 BISPSW—Bit Set PSW

BISPSW (MASKADDR)

Parameters:

MASKADDR
Address of a word whose contents are to be ORed into the PSW.

Result:
NOVALUE

4.11 BPT—Break Point Trap

BPT ( )

Parameters:

None

Result:
NOVALUE

4.12 BUGL—Bugcheck with Long Operand

BUGL (ARG)

Parameters:

ARG
Link-time constant expression (LTCE) to be interpreted by the bugcheck exception-handling code.

Result:
NOVALUE

The following example is based on the DISPAT module in the F11ACP:

```
builtin
    BUGW
external literal
    BUG$_UNXSIGNAL;
......
BUGW( BUG$_UNXSIGNAL or 4 );
```

generates the code sequence:

```
.EXTRN BUG$_UNXSIGNAL
......
BUGW
.WORD BUG$_UNXSIGNAL!4
```

The inline generation of this code eliminates the need to create a PLIT containing hand-assembled code.

4.13 BUGW—Bugcheck with Word Operand

```
BUGW  ARG
```

Parameters:

ARG
Link-time constant expression (LTCE) word value to be interpreted by the bugcheck exception-handling code.

Result:
NOVALUE

4.14 CALLG—Call with General Parameter List

```
CALLG   (ARGLIST, RTN [; OUT0, OUT1])
```

Input Parameters:

ARGLIST
Address of the parameter list to be placed in the address pointer (AP) register (must not be a register name or a %REF value).

RTN
Address of the routine to be called.

Output Parameters:

OUT0
Value returned in R0.

OUT1
Value returned in R1.

Result:
Same as result of the routine that is called.

Note that this function does not interact with any linkage attribute information that may be associated with the second parameter. It must be used only to call a routine with a standard VMS linkage (BLISS and FORTRAN in BLISS–32).

To pass a routine's argument list to another routine, use the following sequence:

```
BUILTIN
    AP,
    CALLG;
CALLG(.AP,OTHERRTN);
```

4.15 CHMx—Change Mode

CHME (ARG)

CHMK (ARG)

CHMS (ARG)

CHMU (ARG)

Parameters:

ARG
Address of a word whose contents are used as a parameter code.

Result:
NOVALUE

4.16 CMPC3—Compare Characters 3 Operand

CMP3 (LENA, SRC1A, SRC2A [; OUT0, OUT1, OUT2, OUT3]))

Input Parameters:

LENA
Address of a word containing the byte comparison lengths for string 1 and string 2.

SRC1A
Address of the base of string 1.

SRC2A
Address of the base of string 2.

Output Parameters:

OUT0
Value returned in R0 is the number of bytes remaining in string 1 (including the byte that terminated the comparison); R0 is zero only if the strings are equal.

OUT1
Value returned in R1 is the address of the byte in string 1 that terminated the comparison; if the strings are equal, R0 contains an address equal to one byte beyond string 1.

OUT2
Value returned in R2 equals R0.

OUT3
Value returned in R3 is the address of the byte in string 2 that terminated the comparison; if the strings are equal, R3 contains an address equal to one byte beyond string 2.

Result:

–1
SRC1A less than SRC2A.

0
SRC1A equal to SRC2A.

1
SRC1A greater than SRC2A.

4.17 CMPC5—Compare Characters 5 Operand

CMPC5 (SRC1LENA, SRC1A, FILLA, SRC2LENA, SRC2A [;OUT0, OUT1, OUT2, OUT3])

Input Parameters:

SRC1LENA
Address of a word containing the length of string 1.

SRC1A
Address of the base of string 1.

FILLA
Address of a word containing a byte fill value, which extends the shorter string to the length of the longer.

SRC2LENA
Address of a word containing the length of string 2.

SRC2A
Address of the base of string 2.

Output Parameters:

OUT0
Value returned in R0 is the number of bytes remaining in string 1 (including the byte which terminated the comparison); R0 is zero if the strings are of equal value and length or if string 1 was exhausted before the comparison terminated.

OUT1
Value returned in R1 is the address of the byte that terminated the comparison; if the comparison did not terminate before string 1 was exhausted, R1 contains an address equal to one byte beyond string 1.

OUT2
Value returned in R2 is the number of bytes remaining in string 2 (including the byte that terminated the comparison); R0 is zero if the strings are of equal value and length or if string 2 was exhausted before the comparison terminated.

OUT3
Value returned in R3 is the address of the byte in string 2 that terminated the comparision; if the comparison did not terminate before string 2 was exhausted, R3 contains an address equal to one byte beyond string 2.

Result:

–1
SRC1A less than SRC2A.

0
SRC1A equal to SRC2A.

1
SRC1A greater than SRC2A.

4.18 CMPD—Compare D-Floating Operands

CMPD (SRC1A, SRC2A)

Parameters:

SRC1A
Address of a quadword containing a double-precision floating-point value (must not be the name of a register nor a %REF value).

SRC2A
Address of a quadword containing a double-precision floating-point value.

Result:

–1
SRC1A less than SRC2A.

0
SRC1A equal to SRC2A.

1
SRC1A greater than SRC2A.

4.19 CMPF—Compare F-Floating Operands

CMPF (SRC1A, SRC2A)

Parameters:

SRC1A
Address of a longword containing a single-precision floating-point value.

SRC2A
Address of a longword containing a single-precision floating-point value.

Result:

–1
SRC1A less than SRC2A.

0
SRC1A equal to SRC2A.

1
SRC1A greater than SRC2A.

4.20 CMPG—Compare G-Floating Operands

CMPG (SRC1A, SRC2A)

Parameters:

SRC1A
Address of a quadword containing an extended double-precision floating-point value.

SRC2A
Address of a quadword containing an extended double-precision floating-point value.

Result:

–1
SRC1A less than SRC2A.

0
SRC1A equal to SRC2A.

1
SRC1A greater than SRC2A.

4.21 CMPH—Compare H-Floating Operands

CMPH (SRC1A, SRC2A)

Parameters:

SRC1A
Address of an octaword containing an extended-exponent double-precision floating-point value.

SRC2A
Address of an octaword containing an extended-exponent double-precision floating-point value.

Result:

–1
SRC1A less than SRC2A.

0
SRC1A equal to SRC2A.

1
SRC1A greater than SRC2A.

4.22 CMPM—Compare Multiword Operands

CMPP (SIZE, SRC1A, SRC2A)

Parameters:

SIZE
Compile-time-constant expression indicating the size of the operands in longwords (BLISS value units).

SRC1A
Address of a longword containing an extended multiprecision value.

SRC2A
Address of a longword containing an extended multiprecision value.

Result:

–1
SRC1A less than SRC2A.

0
SRC1A equal to SRC2A.

1
SRC1A greater than SRC2A.

4.23 CMPP—Compare Packed

CMPP (SRC1LENA, SRC1ADDR, SRC2LENA, SRC2ADDR [; OUT0, OUT1, OUT2, OUT3])

Input Parameters:

SRC1LENA
Address of a word containing the length of the decimal string SRC1.

SRC1ADDR
Address of the base of packed decimal string SRC1.

SRC2LENA
Address of a word containing the length of decimal string SRC2.

SRC2ADDR
Address of the base of packed decimal string SRC2.

Output Parameters:

OUT0
Value returned in R0 is zero.

OUT1
Value returned in R1 is the address of the byte containing the most significant digit of string SRC1.

OUT2
Value returned in R2 is zero.

OUT3
Value returned in R3 is the address of the byte containing the most significant digit of string SRC2.

Result:

–1
SRC1 less than SRC2.

0
SRC1 equal to SRC2.

1
SRC1 greater than SRC2.

Note that CMPP3 or CMPP4 is generated depending on the operands provided.

4.24 CRC—Cyclic Redundancy Check

CRC (TABLEADDR, INICRCADDR, STRLENADDR, STREAMADDR, DSTADDR)

Parameters:

TABLEADDR
Address of a 16-longword table.

INICRCADDR
Address of a longword which contains the initial CRC.

STRLENADDR
Address of a word containing the unsigned length of the data stream in bytes.

STREAMADDR
Address of the first byte of the data stream.

DSTADDR
Address of a longword where the resulting 32-bit CRC is to be stored.

Result:
NOVALUE

4.25 CVTDF—Convert D-Floating to F-Floating

CVTDF (SRCA, DSTA)

Parameters:
SRCA
Address of a quadword containing a double-precision floating-point value.

DSTA
Address of a longword where the single-precision floating-point conversion is stored.

Result:
1
No floating-point overflow.

0
Floating-point overflow.

4.26 CVTDI—Convert D-Floating to Integer

CVTDI (SRCA, DSTA)

Parameters:
SRCA
Address of a quadword containing a double-precision floating-point value.

DSTA
Address where the integer conversion is stored.

Result:
1
No integer overflow.

0
Integer overflow.

4.27 CVTDL—Convert D-Floating to Long

CVTDL (SRCA, DSTA)

Parameters:

SRCA
Address of a quadword containing a double-precision floating-point value.

DSTA
Address of a longword where the integer conversion is stored.

Result:

1
No integer overflow.

0
Integer overflow.

4.28 CVTFD—Convert F-Floating to D-Floating

CVTFD (SRCA, DSTA)

Parameters:

SRCA
Address of a longword containing a single-precision floating-point value.

DSTA
Address of a quadword where the double-precision floating-point conversion is stored.

Result:
NOVALUE

4.29 CVTFG—Convert F-Floating to G-Floating

CVTFG (SRCA, DSTA)

Parameters:

SRCA
Address of a longword containing a single-precision floating-point value.

DSTA
Address of a quadword where the extended double-precision floating-point conversion is stored.

Result:
NOVALUE

4.30 CVTFH—Convert F-Floating to H-Floating

CVTFH (SRCA, DSTA)

Parameters:

SRCA
Address of a longword containing a single-precision floating-point value.

DSTA
Address of an octaword where the extended-exponent double-precision floating-point conversion is stored.

Result:
NOVALUE

4.31 CVTFI—Convert F-Floating to Integer

CVTFI (SRCA, DSTA)

Parameters:

SRCA
Address of a longword containing a single-precision floating-point value.

DSTA
Address of a longword where the integer conversion is stored.

Result:

1
No integer overflow.

0
Integer overflow.

4.32 CVTFL—Convert F-Floating to Long

CVTFL (SRCA, DSTA)

Parameters:

SRCA
Address of a longword containing a single-precision floating-point value.

DSTA
Address of a longword where the integer conversion is stored.

Result:

1
No integer overflow.

0
Integer overflow.

4.33 CVTGF—Convert G-Floating to F-Floating

CVTGF (SRCA, DSTA)

Parameters:

SRCA
Address of a quadword containing an extended double-precision floating-point value.

DSTA
Address of a longword where the single-precision floating-point conversion is stored.

Result:
NOVALUE

4.34 CVTGH—Convert G-Floating to H-Floating

CVTGH (SRCA, DSTA)

Parameters:

SRCA
Address of a quadword containing an extended double-precision floating-point value.

DSTA
Address of an octaword where the extended-exponent double-precision floating-point conversion is stored.

Result:
NOVALUE

4.35 CVTGL—Convert G-Floating to Long

CVTGL (SRCA, DSTA)

Parameters:

SRCA
Address of a quadword containing an extended double-precision floating-point value.

DSTA
Address of a longword where the integer conversion is stored.

Result:
NOVALUE

4.36 CVTHF—Convert H-Floating to F-Floating

CVTHF (SRCA,DSTA)

Parameters:

SRCA
Address of an octaword containing an extended-exponent double-precision floating-point value.

DSTA
Address of a longword where the single-precision floating-point conversion is stored.

Result:
NOVALUE

4.37 CVTHG—Convert H-Floating to G-Floating

CVTHG (SRCA, DSTA)

Parameters:

SRCA
Address of an octaword containing an extended-exponent double-precision floating-point value.

DSTA
Address of a quadword where the extended double-precision floating-point conversion is stored.

4.38 CVTHL—Convert H-Floating to Long

CVTHG (SRCA, DSTA)

Parameters:

SRCA
Address of an octaword containing an extended-exponent double-precision floating-point value.

DSTA
Address of a longword where the integer conversion is stored.

Result:
NOVALUE

4.39 CVTID—Convert Integer to D-Floating

CVTID (SRCA, DSTA)

Parameters:

SRCA
Address of a longword containing an integer value.

DSTA
Address of a quadword where the double-precision floating-point conversion is stored.

Result:
NOVALUE

4.40 CVTIF—Convert Integer to F-Floating

CVTIF (SRCA, DSTA)

Parameters:

SRCA
Address of a longword containing an integer value.

DSTA
Address of a longword where the single-precision floating-point conversion is stored.

Result:
NOVALUE

4.41 CVTLD—Convert Long to D-Floating

CVTLD (SRCA, DSTA)

Parameters:

SRCA
Address of a longword containing an integer value.

DSTA
Address of a quadword where the double-precision floating-point conversion is stored.

Result:
NOVALUE

No overflow can occur.

4.42 CVTLF—Convert Long to F-Floating

CVTLF (SRCA, DSTA)

Parameters:

SRCA
Address of a longword containing an integer value.

DSTA
Address of a longword where the single-precision floating-point conversion is stored.

Result:
NOVALUE

No overflow can occur.

4.43 CVTLH—Convert Long to H-Floating

CVTLH (SRCA, DSTA)

Parameters:

SRCA
Address of a longword containing an integer value.

DSTA
Address of an octaword where the extended-exponent double-precision floating-point conversion is stored.

Result:
NOVALUE

4.44 CVTLP—Convert Long to Packed

CVTLP (SRCA, DSTLENA, DSTADDR)

Parameters:

SRCA
Address of a longword containing an integer value.

DSTLENA
Address of a word containing the length of the destination string.

DSTADDR
Address of the base of the destination string.

Result:

1
No decimal overflow.

0
Decimal overflow.

4.45 CVTPL—Convert Packed to Long

CVTPL (SRCLENA, SRCADDR, DSTA)

Parameters:

SRCLENA
Address of a word containing the length of the source string.

SRCADDR
Address of the base of the source string.

DSTA
Address of a longword where the integer conversion is stored.

Result:

1
No integer overflow.

0
Integer overflow.

4.46 CVTPS—Convert Packed to Leading Separate Numeric

CVTPS (SRCLENA, SRCADDR, DSTLENA, DSTADDR [; OUT0, OUT1, OUT2, OUT3])

Input Parameters:

SRCLENA
Address of a word containing the length of the source string.

SRCADDR
Address of the base of the source string.

DSTLENA
Address of a word containing the length of the destination string.

DSTADDR
Address of the base of the destination string.

Output Parameters:

OUT0
Value returned in R0 is zero.

OUT1
Value returned in R1 is the address of the byte containing the most significant digit of the source string.

OUT2
Value returned in R2 is zero.

OUT3
Value returned in R3 is the address of the sign byte of the destination string.

Result:

1
No decimal overflow.

0
Decimal overflow.

4.47 CVTPT—Convert Packed to Trailing Numeric

CVTPS (SRCLENA, SRCADDR, TBLADDR, DSTLENA, DSTADDR [; OUT0, OUT1, OUT2, OUT3])

Input Parameters:

SRCLENA
Address of a word containing the length of the source string.

SRCADDR
Address of the base of the source string.

TBLADDR
Address of the table used to convert the sign.

DSTLENA
Address of a word containing the length of the destination string.

DSTADDR
Address of the base of the destination string.

Output Parameters:

OUT0
Value returned in R0 is zero.

OUT1
Value returned in R1 is the address of the byte containing the most significant digit of the source string.

OUT2
Value returned in R2 is zero.

OUT3
Value returned in R3 is the address of the sign byte of the destination string.

Result:

1
No decimal overflow.

0
Decimal overflow.

4.48 CVTRDL—Convert Rounded D-Floating to Long

CVTRDL (SRCA, DSTA)

Parameters:

SRCA
Address of a quadword containing a double-precision floating-point value.

DSTA
Address of a longword where the integer conversion is stored.

Result:

1
No integer overflow.

0
Integer overflow.

4.49 CVTRFL—Convert Rounded F-Floating to Long

CVTRFL (SRCA, DSTA)

Parameters:

SRCA
Address of a longword containing a single-precision floating-point value.

DSTA
Address of a longword where the integer conversion is stored.

Result:

1
No integer overflow.

0
Integer overflow.

4.50 CVTRGL—Convert Rounded G-Floating to Long

CVTRGL (SRCA, DSTA)

Parameters:

SRCA
Address of a quadword containing an extended double-precision floating-point value.

DSTA
Address of a longword where the integer conversion is stored.

Result:

1
No integer overflow.

0
Integer overflow.

4.51 CVTRHL—Convert Rounded H-Floating to Long

CVTRHL (SRCA, DSTA)

Parameters:

SRCA
Address of an octaword containing an extended-exponent double-precision floating-point value.

DSTA
Address of a longword where the integer conversion is stored.

Results:

1
No integer overflow.

0
Integer overflow.

4.52 CVTSP—Convert Leading Separate to Packed

CVTSP (SRCLENA, SRCADDR, DSTLENA, DSTADDR [;OUT0, OUT1, OUT2, OUT3])

Input Parameters:

SRCLENA
Address of a word containing the length of the source string.

SRCADDR
Address of the base of the source string.

DSTLENA
Address of a word containing the length of the destination string.

DSTADDR
Address of the base of the destination string.

Output Parameters:

OUT0
Value returned in R0 is zero.

OUT1
Value returned in R1 is the address of the sign byte of the source string.

OUT2
Value returned in R2 is zero.

OUT3
Value returned in R3 is the address of the byte containing the most significant digit of the destination string.

Result:

1
No decimal overflow.

0
Decimal overflow.

4.53 CVTTP—Convert Trailing Numeric to Packed

CVTTP (SRCLENA, SRCADDR, TBLADDR, DSTLENA, DSTADDR [; OUT0, OUT1, OUT2, OUT3])

Input Parameters:

SRCLENA
Address of a word containing the length of the source string.

SRCADDR
Address of the base of the source string.

TBLADDR
Address of the table used to convert the sign.

DSTLENA
Address of a word containing the length of the destination string.

DSTADDR
Address of the base of the destination string.

Output Parameters:

OUT0
Value returned in R0 is zero.

OUT1
Value returned in R1 is the address of the most significant digit of the source string.

OUT2
Value returned in R2 is zero.

OUT3
Value returned in R3 is the address of the byte containing the most significant digit of the destination string.

Result:

1
No decimal overflow.

0
Decimal overflow.

4.54 DIVD—Divide D-Floating Operands

DIVD (DIVSR, DIVID, QUOT)

Parameters:

DIVSR
Address of a double-precision floating-point quadword used as the divisor.

DIVID
Address of a double-precision floating-point quadword used as the dividend.

QUOT
Address of a quadword where the quotient is stored.

Result:
NOVALUE

4.55 DIVF—Divide F-Floating Operands

DIVF (DVISR, DIVID, QUOT)

Parameters:

DVISR
Address of a single-precision floating-point longword used as the divisor.

DIVID
Address of a single-precision floating-point longword used as the dividend.

QUOT
Address of a longword where the quotient is stored.

Result:
NOVALUE

4.56 DIVG—Divide G-Floating Operands

DIVG (DVISR, DIVID, QUOT)

Parameters:

DIVSR
Address of an extended double-precision floating-point quadword used as the divisor.

DIVID
Address of an extended double-precision floating-point quadword used as the dividend.

QUOT
Address of a quadword where the quotient is stored.

Result:
NOVALUE

4.57 DIVH—Divide H-Floating Operands

DIVH (DIVSR, DIVID, QUOT)

Parameters:

DIVSR
Address of an extended-exponent double-precision floating-point octaword that is used as the divisor.

DIVID
Address of an extended-exponent double-precision floating-point octaword that is used as the dividend.

QUOT
Address of an octaword where the quotient is stored.

Result:
NOVALUE

4.58 EDITPC—Edit Packed to Character

EDITPC (SRCLENA, SRCADDR, PATTERN, DSTADDR [; OUT0, OUT1, OUT2, OUT3, OUT4, OUT5])

Input Parameters:

SRCLENA
Address of a word containing the length of the source string.

SRCADDR
Address of the base of the source string.

PATTERN
Address of the pattern-operator string.

DSTADDR
Address of the base of the destination string.

Output Parameters:

OUT0
Value returned in R0 is the length of the source string.

OUT1
Value returned in R1 is the address of the byte containing the most significant digit of the source string.

OUT2
Value returned in R2 is zero.

OUT3
Value returned in R3 is the address of the byte containing the EO$END pattern operator.

OUT4
Value returned in R4 is zero.

OUT5
Value returned in R5 is the address of one byte beyond the last byte of the destination string.

Result:

1
No decimal overflow.

0
Decimal overflow.

4.59 EDIV—Extended-Precision Divide

EDIV (DIVISOR, DIVIDEND, QUOTIENT, REMAINDER)

Parameters:

DIVISOR
Address of a longword whose contents are used as the divisor.

DIVIDEND
Address of a quadword whose contents are used as the dividend.

QUOTIENT
Address of a longword where the quotient is to be stored.

REMAINDER
Address of a longword where the remainder is to be stored.

Result:

Contents of the PSL

4.60 EMUL—Extended-Precision Multiply

EMUL (MULR, MULD, ADD, PROD)

Parameters:

MULR
Address of a longword whose contents are used as the multiplier.

MULD
Address of a longword whose contents are used as the multiplicand.

ADD
Address of a longword that is sign-extended to a quadword and added to the product quadword.

PROD
Address of a quadword where the result is to be stored.

Result:

Contents of the PSL

4.61 FFC and FFS—Find and Modify Operations

FFC (POSADDR, SIZADDR, BASEADDR, DSTADDR) Find first clear bit

FFS (POSADDR, SIZADDR, BASEADDR, DSTADDR) Find first set bit

Parameters:

POSADDR
Address of a longword whose contents specify a bit position relative to the low bit of the byte addressed by BASEADDR. The low bit of that byte is bit position zero.

SIZADDR
Address of a byte whose content, when zero extended to 32 bits, has a value less than or equal to 32. This value specifies the size of the field to be searched. The size of the field is measured in bits, begins at the bit position specified by POSADDR, and extends toward increasing bit numbers.

BASEADDR
Address of a byte whose low bit is interpreted as position zero.

DSTADDR
Address of a longword where the position of the first bit found in the specified state is to be stored.

Result:

1
No bit in the specified state is found in the field searched.

0
Otherwise.

4.62 HALT—Halt Processor

HALT

Parameters:
None

Result:
NOVALUE

Description:
This routine generates a HALT instruction.

4.63 INDEX—Index Calculation

INDEX (SUBSCRIPT, LOW, HIGH, SIZE, INDEXIN, INDEXOUT)

Parameters:

SUBSCRIPT
Address of a longword containing the subscript.

LOW
Address of a longword containing the lower bound of the subscript range.

HIGH
Address of a longword containing the upper bound of the subscript range.

SIZE
Address of a longword containing the scale factor.

INDEXIN
Address of a longword containing the initial index value.

INDEXOUT
Address of a longword where the result is to be stored.

Result:
NOVALUE

4.64 INSQHI and INSQTI—Insert Entry in Queue, Interlocked

INSQHI(ENTRY, HEADER)   Insert entry in queue at head, interlocked

INSQTI(ENTRY, HEADER)   Insert entry in queue at tail, interlocked

Parameters:

ENTRY
Address of an entry to be inserted in a queue following the header.

HEADER
Address of the queue header.

Result:

0
If ENTRY was not the first entry to be inserted in the queue.

1
If the secondary interlock could not be acquired.

2
If ENTRY was the first entry to be inserted into the queue.

Note that if a real result is needed, the instruction sequence generated for these functions is as follows:

```
      CLRL    tmp
      INSQxI  ENTRY,HEADER
      BCS     1$
      BNEQ    2$
      INCL    tmp
 1$:  INCL    tmp
 2$:
```

If only a flow result is requested, the instruction sequence is as follows:

```
      INSQxI   ENTRY,HEADER
      BCC      1$          ; Success
      ....                 ; Failure actions
```

Thus, the BLISS expression to set a software interlock realized with a queue is as follows:

```
 WHILE INSQHI(ENTRY, HEADER ) DO WAIT();
```

4.65 INSQUE—Insert Entry in Queue

INSQUE   (ENTRY, PRED)

Parameters:

ENTRY
Address of an entry to be inserted in the queue after the entry specified by PRED.

PRED
Address of an entry in a queue.

Result:

1
ENTRY was the first entry to be inserted into the queue.

0
Otherwise.

4.66 LOCC—Locate Character

LOCC (CHARA, LENA, ADDR [; OUT0, OUT1])

Input Parameters:

CHARA
Address of a byte containing the character to be located.

LENA
Address of a word containing the search string length.

ADDR
Address of the string to be searched.

Output Parameters:

OUT0
Value returned in R0 is the number of bytes remaining in the string (including the located one) if the byte is located; otherwise, R0 is a zero.

OUT1
Value returned in R1 is the address of the byte located if the byte is located; otherwise, R1 is the address of one byte beyond the string.

Result:

1
A byte was located which was the same as .CHARA<0,8> (the Z condition code is clear).

0
No byte was located (the Z condition code is set).

4.67 MATCHC—Match Characters

MATCHC (OBJLENA, OBJADDR, SRCLENA, SRCADDR [; OUT0, OUT1, OUT2, OUT3])

Input Parameters:

OBJLENA
Address of word containing the length of the pattern string.

OBJADDR
Address of the pattern string.

SRCLENA
Address of a word containing the length of the string to be searched.

SRCADDR
Address of the string to be searched.

Output Parameters:

OUT0
Value returned in R0 is zero (if a match occurred); otherwise, R0 is the number of bytes in the object string.

OUT1
Value returned in R1 is the address of one byte beyond the object string (if a match occurred); otherwise, it is the address of the object string.

OUT2
Value returned in R2 is the number of bytes remaining in the source string after the match (if a match occurred); otherwise, R2 is zero.

OUT3
Value returned in R3 is the address of one byte beyond the last byte matched (if a match occurred); otherwise, R3 is the address of one byte beyond the source string.

Result:

1
A match was found (the Z condition code is set).

0
No match was found (the Z condition code is clear).

4.68 MFPR—Move from Processor Register

MFPR (PROCREG, DSTADDR)

Parameters:

PROCREG
Processor register number.

DSTADDR
Address of a longword where the contents of the processor register is to be stored.

Result:
NOVALUE

4.69 MOVC3—Move Character 3 Operand

MOVC3 (LENA, SRCADDR, DSTADDR [; OUT0, OUT1, OUT2, OUT3, OUT4, OUT5])

Input Parameters:

LENA
Address of a word containing the length of the source string.

SRCADDR
Address of the base of the source string.

DSTADDR
Address of the destination string.

Output Parameters:

OUT0
Value returned in R0 is zero.

OUT1
Value returned in R1 is the address of one byte beyond the source string.

OUT2
Value returned in R2 is zero.

OUT3
Value returned in R3 is the address of one byte beyond the destination string.

OUT4
Value returned in R4 is zero.

OUT5
Value returned in R5 is zero.

Result:
NOVALUE

4.70 MOVC5—Move Character 5 Operand

MOVC5 (SRCLENA, SRCADDR, FILL, DSTLENA, DSTADDR [; OUT0, OUT1, OUT2, OUT3, OUT4, OUT5])

Input Parameters:

SRCLENA
Address of word containing the length of the source string.

SRCADDR
Address of the base of the source string.

FILL
Address of a byte containing the fill character.

DSTLENA
Address of word containing the destination-string length.

DSTADDR
Address of the destination string.

Output Parameters:

OUT0
Value returned in R0 is the number of unmoved bytes remaining in the source string; R0 is nonzero only if the source string is longer than the destination string.

OUT1
Value returned in R1 is the address of one byte beyond the last byte in the source string that was moved.

OUT2
Value returned in R2 is zero.

OUT3
Value returned in R3 is the address of one byte beyond the destination string.

OUT4
Value returned in R4 is zero.

OUT5
Value returned in R5 is zero.

Result:
NOVALUE

4.71 MOVP—Move Packed

MOVP (LENA, SRCADDR, DSTADDR [; OUT0, OUT1, OUT2, OUT3])

Input Parameters:

LENA
Address of a word containing the length of the source string.

SRCADDR
Address of the base of the source decimal string.

DSTADDR
Address of the base of the destination.

Output Parameters:

OUT0
Value returned in R0 is zero.

OUT1
Value returned in R1 is the address of the byte containing the most significant digit of the source string.

OUT2
Value returned in R2 is zero.

OUT3
Value returned in R3 is the address of the byte containing the most significant digit of the destination string.

Result:
NOVALUE

4.72 MOVPSL—Move from PSL

MOVPSL (DSTADDR)

Parameters:

DSTADDR
Address of a longword where the contents of the PSL are to be stored.

Result:
NOVALUE

4.73 MOVTC—Move Translated Characters

MOVTC (SRCLENA, SRCADDR, FILL, TBLADDR, DSTLENA, DSTADDR [; OUT0, OUT1, OUT2, OUT3, OUT4, OUT5])

Input Parameters:

SRCLENA
Address of a word containing the source length.

SRCADDR
Address of source string.

FILL
Address of a byte containing the fill character.

TBLADDR
Address of the translation table.

DSTLENA
Address of a word containing the destination length.

DSTADDR
Address of the destination string.

Output Parameters:

OUT0
Value returned in R0 is the number of translated bytes remaining in the source string; R0 is nonzero only if the source string is longer than the destination string.

OUT1
Value returned in R1 is the address of one byte beyond the last byte in the source string that was translated.

OUT2
Value returned in R2 is zero.

OUT3
Value returned in R3 is the address of the translation table.

OUT4
Value returned in R4 is zero.

OUT5
Value returned in R5 is the address of one byte beyond the destination string.

Result:
NOVALUE

4.74 MOVTUC—Move Translated Until Character

MOVTUC (SRCLENA, SRCADDR, ESCADDR, TBLADDR, DSTLENA, DSTADDR [; OUT0, OUT1, OUT2, OUT3, OUT4, OUT5])

Input Parameters:

SRCLENA
Address of a word containing the length of the source string.

SRCADDR
Address of the base of the source string.

ESCADDR
Address of a byte containing the escape character.

TBLADDR
Address of the translation table (can be created with CH$TRANSTABLE).

DSTLENA
Address of a word containing the length of the destination string.

DSTADDR
Address of the base of the destination string.

Output Parameters:

OUT0
Value returned in R0 is the number of bytes remaining in the source string (including the byte that caused the escape); R0 is zero only if the entire source string was translated and moved without escape.

OUT1
Value returned in R1 is the address of the byte that resulted in destination string exhaustion or escape; if no exhaustion or escape, R1 is the address of one byte beyond the end of the source string.

OUT2
Value returned in R2 is zero.

OUT3
Value returned in R3 is the address of the table.

OUT4
Value returned in R4 is the number of bytes remaining in the destination string.

OUT5
Value returned in R5 is the address of the byte in the destination string that would have received the translated byte that caused the escape, or would have received a translated byte if the source string was not exhausted; if no exhaustion or escape, R5 is the address of one byte beyond the end of the destination string.

Result:

If the string was successfully translated without escape, the result is 0. Otherwise, the result is the address of the byte in the source string which caused the escape.

4.75 MTPR—Move to Processor Register

MTPR (SRCADDR, PROCREG)

Parameters:

SRCADDR
Address of a longword whose contents are to be loaded into the designated processor register.

PROCREG
Processor register number.

Result:
NOVALUE

4.76 MULD—Multiply D-Floating Operands

MULD (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A
Address of a double-precision floating-point quadword used as the multiplier.

SRC2A
Address of a double-precision floating-point quadword used as the multiplicand.

DSTA
Address of a quadword where the product is stored.

Result:
NOVALUE

4.77 MULF—Multiply F-Floating Operands

MULF (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A
Address of a single-precision floating-point longword used as the multiplier.

SRC2A
Address of a single-precision floating-point longword used as the multiplicand.

DSTA
Address of a longword where the product is stored.

Result:
NOVALUE

4.78 MULG—Multiply G-Floating Operands

MULG (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A
Address of an extended double-precision floating-point quadword used as the multiplier.

SRC2A
Address of an extended double-precision floating-point quadword used as the multiplicand.

DSTA
Address of a quadword where the product is stored.

Result:
NOVALUE

4.79 MULH—Multiply H-Floating Operands

MULH (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A
Address of an extended-exponent double-precision floating-point octaword used as a multiplier.

SRC2A
Address of an extended-exponent double-precision floating-point octaword used as a multiplicand.

DSTA
Address of an octaword where the product is stored.

Result:
NOVALUE

4.80 NOP—No Operation

NOP ( )

Parameters:

None

Result:
NOVALUE

4.81 PROBER—Probe Read Accessibility

PROBER (MODEADDR, LENGTHADDR, BASEADDR)

Parameters:

MODEADDR
Address of a byte whose contents specify a protection mode.

LENGTHADDR
Address of a word whose contents are zero extended decremented by 1 and added to BASEADDR to specify the last byte of an area in memory.

BASEADDR
Address of the first byte of an area in memory (must not be the name of a register nor a %REF value).

Result:

1
Both first and last bytes are read accessible.

0
Otherwise.

4.82 PROBEW—Probe Write Accessibility

PROBEW (MODEADDR, LENGTHADDR, BASEADDR)

Parameters:

MODEADDR
Address of a byte whose contents specify a processor mode.

LENGTHADDR
Address of a word whose contents are zero extended decremented by 1 and added to BASEADDR to specify the last byte of an area in memory.

BASEADDR
Address of the first byte of an area in memory (must not be the name of a register nor a %REF value).

Result:

1
Both first and last bytes are write accessible.

0
Otherwise.

4.83 REMQHI and REMQTI—Remove Entry From Queue, Interlocked

REMQHI (HEADER, ADDR) Remove from queue at head, interlocked

REMQTI (HEADER, ADDR) Remove from queue at tail, interlocked

Parameters:

HEADER
Address of the queue header.

ADDR
Address of a longword where the address of the entry removed is to be stored.

Result:

0
The queue is not empty; an entry was removed.

1
The secondary interlock could not be acquired.

2
The queue is now empty; the last entry was removed.

3
The queue was already empty; no entry was removed.

The result of REMQxI is TRUE if no entry was removed, either because the queue was empty or because the secondary interlock was not acquired.

The instruction sequence generated in real-context is as follows:

```
     CLRL     tmp
     REMQxI   HEADER,ADDR
     BCS      2$           ; Interlock failed
     BVS      1$           ; Remove failed
     BEQL     3$           ; Removed an entry
     DECL     tmp          ; Removed last entry
 1$: ADDL     #2,tmp
 2$: INCL     tmp
```

The instruction sequence generated in flow-context is as follows:

```
     REMQxI   HEADER,ADDR
     BVC      1$           ; Entry removed
     .....                 ; Remove failed
```

4.84 REMQUE—Remove Entry from Queue

REMQUE (ENTRY, ADDR)

Parameters:

ENTRY
Address of an entry in a queue.

ADDR
Address of a longword where the address of the entry removed is to be stored.

Result:

Value	Initial State	Final State	Entry Removed
0	Not empty	Not empty	Yes
2	Not empty	Empty	Yes
3	Empty	Empty	No

Note that the value of REMQUE is true only if no entry was removed.

The value is computed from the condition codes as follows:

```
(Z-bit)^1 OR (V-bit)
```

4.85 ROT—Rotate a Value

ROT (VALUE, SHIFT)

Parameters:

VALUE
Value to be rotated.

SHIFT
Number of bits to rotate.

Result:
VALUE rotated the specified number of bits.

4.86 SCANC—Scan Characters

SCANC (LENA, ADDR, TBLADDR, MASKA [; OUT0, OUT1, OUT2, OUT3])

Input Parameters:

LENA
Address of a word containing the length of the string to be scanned.

ADDR
Address of the base of the string.

TBLADDR
Address of the translation table.

MASKA
Address of a byte containing the mask to use in scanning.

Output Parameters:

OUT0
Value returned in R0 is the number of bytes remaining in the string (including the byte that produced a nonzero AND result); R0 is zero only if there was a zero AND result.

OUT1
Value returned in R1 is the address of the byte that produced a nonzero AND result or (if no nonzero AND result) the address of one byte beyond the string.

OUT2
Value returned in R2 is zero.

OUT3
Value returned in R3 is the address of the table.

Result:

If the SCANC fails to find a match, the result is 0. Otherwise, the result is the address of the byte that produced a nonzero AND with the MASK.

4.87 SKPC—Skip Character

SKPC (CHARA, LENA, ADDR [; OUT0, OUT1]))

Input Parameters:

CHARA
Address of a byte containing the character to be skipped.

LENA
Address of a word containing the search string length.

ADDR
Address of the string to be searched.

Output Parameters:

OUT0
Value returned in R0 is the number of bytes remaining in the string (including the located or unequal one) if the (unequal) byte is located; otherwise, R0 is zero.

OUT1
Value returned in R1 is the address of the byte located (if byte is located); otherwise, R1 is the address of one byte beyond the string.

Result:

1
A byte was located which was not the same as .CHARA<0,8> (the Z condition code is clear).

0
No byte was located (the Z condition code is set).

4.88 SPANC—Span Characters

SPANC (LENA, ADDR, TBLADDR, MASKA [; OUT0, OUT1, OUT2, OUT3])

Input Parameters:

LENA
Address of a word containing the length of the string to be spanned.

ADDR
Address of the base of the string.

TBLADDR
Address of the translation table.

MASKA
Address of a byte containing the mask to use in spanning.

Output Parameters:

OUT0
Value returned in R0 is the number of bytes remaining in the string (including the byte that produced a zero AND result); R0 is a zero only if there was a nonzero AND result.

OUT1
Value returned in R1 is the address of the byte that produced a zero AND result; otherwise, R1 is the address of one byte beyond the string.

OUT2
Value returned in R2 is zero.

OUT3
Value returned in R3 is the address of the table.

Result:

If the SPANC fails to find a match, the result is zero. Otherwise, the result is the address of the byte that produced a zero AND with the MASK.

4.89 SUBD—Subtract D-Floating Operands

SUBD (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A
Address of a double-precision floating-point quadword used as the subtrahend.

SRC2A
Address of a double-precision floating-point quadword used as the minuend.

DSTA
Address of the quadword where the difference is stored.

Result:
NOVALUE

4.90 SUBF—Subtract F-Floating Operands

SUBF (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A
Address of a single-precision floating-point longword used as the subtrahend.

SRC2A
Address of a single-precision floating-point longword used as the minuend.

DSTA
Address of a longword where the difference is stored.

Result:
NOVALUE

4.91 SUBG—Subtract G-Floating Operands

SUBG (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A
Address of an extended double-precision floating-point quadword used as the subtrahend.

SRC2A
Address of an extended double-precision floating-point quadword used as the minuend.

DSTA
Address of a quadword where the difference is stored.

Result:
NOVALUE

4.92 SUBH—Subtract H-Floating Operands

SUBH (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A
Address of an extended-exponent double-precision floating-point octaword used as the subtrahend.

SRC2A
Address of an extended-exponent double-precision floating-point octaword used as the minuend.

DSTA
Address of an octaword where the difference is stored.

Result:
NOVALUE

4.93 SUBM—Subtract Multiword Operands

SUBM (SIZE, SRC1A, SRC2A, DSTA)

Parameters:

SIZE
Compile-time-constant-expression indicating the size of the operands in fullwords (BLISS value units).

SRC1A
Address of an extended multiprecision integer used as the subtrahend.

SRC2A
Address of an extended multiprecision integer used as the minuend.

DSTA
Address of the difference.

Result:
NOVALUE

4.94 TESTBITx—Test and Modify Operations

TESTBITSS (FIELD) Test for bit set, then set bit

TESTBITSC (FIELD) Test for bit set, then clear bit

TESTBITCS (FIELD) Test for bit clear, then set bit

TESTBITCC (FIELD) Test for bit clear, then clear bit

TESTBITSSI (FIELD) Test for bit set, then set bit (interlocked)

TESTBITCCI (FIELD) Test for bit clear, then clear bit (interlocked)

Parameters:

FIELD
Address with optional field selector which specifies a field whose low bit will be tested for a particular state. That bit will be set to the specified state. The size parameter of the field selector is ignored, and a literal value one is substituted.

Note that these are the only routines in the machine-specific set which accept an address with a field selector that cannot be evaluated to an address.

Result:

1
Bit tested was in the specified state.

0
Otherwise.

4.95 XFC—Extended Function Call

XFC (OPCODE)

Parameters:

OPCODE
Link-time-constant-expression to be deposited in the byte which follows the XFC opcode in the instruction stream.

Result:

NOVALUE

5
Programming Considerations

This chapter provides practical help on writing BLISS programs. First, usage differences between LIBRARY and REQUIRE files are considered. Then, some common BLISS programming errors are discussed.

5.1 Library and Require File Differences

BLISS library files are used like required files: declarations that are common to more than one module are centralized in a single file, which is automatically incorporated into other modules during compilation by means of REQUIRE or LIBRARY declarations.

Library files are more efficient for doing this than required files for two reasons. First, with files invoked by REQUIRE declarations, the cost of processing the source occurs every time the file is used in a compilation. However, with library files, the major compilation cost occurs once when the library is compiled, and a much smaller cost occurs each time the library file is used in a compilation. A library file closely approximates the internal symbol table representation used by the compiler; hence, the costs of lexical processing (including scanning, lexical conditionals, lexical functions, and macro expansions) and declaration parsing and checking occur only during the library compilation.

Second, with files invoked by REQUIRE declarations, all declarations contained in the file are incorporated into the compiler symbol table. With library files, the compiler does not incorporate declarations into the normal symbol table until they are actually needed. Declarations of names that are not used do not fill up the symbol table.

The difference in cost depends on many factors, including the size of the library, the size of the module being compiled, and the percentage and kind of declarations used from a library. Experimental results indicate that compiler time and space requirements can typically be improved by a factor of 4 by using library files instead of source files.

Library files and declarations and REQUIRE files and declarations are similar; however, the differences are:

- Files invoked by REQUIRE declarations are source (text) files, while files invoked by LIBRARY declarations are special files, previously created by the compiler through a library compilation.
- Files invoked by REQUIRE declarations can contain any source text that is valid when that source text is substituted for the REQUIRE declaration. Files invoked by LIBRARY declarations must be compiled from sources that consist of a sequence of only the following declarations:

    BIND[1]

[1] Some restrictions apply to these declarations.

BIND ROUTINE[1]
BUILTIN[1]
COMPILETIME
EXTERNAL
EXTERNAL LITERAL
EXTERNAL ROUTINE
FIELD
KEYWORDMACRO
LIBRARY
LINKAGE
LITERAL
MACRO
REQUIRE
STRUCTURE
SWITCHES
UNDECLARE

- SWITCHES declarations contained in files invoked by the REQUIRE declaration can affect the module being compiled; those contained in files used to produce library files affect only the library compilation. Switch settings are not incorporated into the compilation that uses the library file.

  (The only switches that are useful in a library compilation are LIST, LANGUAGE, and ERRS. The remaining switches can be given, but are effectively ignored because none of them has any effect on the declarations that are valid in a library compilation.)

- Files invoked by REQUIRE declarations can have effects that depend on previous declarations or switch settings in the module being compiled. This can occur in a lexical conditional (%IF-%THEN-%ELSE-%FI) or macro expansion that depends either on the lexical functions %SWITCHES, %DECLARED, or %VARIANT, or on values of predeclared literals, such as %BPVAL (refer to Section 6.3.1.1). Files invoked by LIBRARY declarations do not have effects that depend on previous declarations or switch settings, because SWITCHES declarations, REQUIRE declarations, lexical conditionals, or macro calls (contained in sources used to produce a library file) are processed during the library compilation.

- Appropriately written source files can be invoked by REQUIRE declarations in BLISS compilers other than BLISS–32. Library files can be invoked only in compilations by the same compiler that created the library file.

In most cases, the source files used to create a library can be invoked by a REQUIRE declaration or the library can be invoked by a LIBRARY declaration, achieving identical effects in the module being compiled. However, the differences presented above can lead to problems that are difficult to identify. Therefore, you should use one consistent form for each set of declarations.

5.2 Frequent BLISS Coding Errors

Certain coding errors occur frequently, especially when new BLISS users compile and debug a new module. The following check list may be useful when you cannot seem to find the source of a problem.

5.2.1 Missing Dots

The most frequent error is to forget a dot. Except for the left side of an assignment expression, the appearance of a data segment name without a preceding fetch operator is the exception, and usually a mistake. For example:

```
IF A THEN ...
```

This should almost certainly be the following:

```
IF .A THEN ...
```

5.2.2 Valued and Nonvalued Routines

The BLISS compiler does not diagnose useless value expressions in contexts that do not use a value. For example:

```
ROUTINE R(A): NOVALUE =
  BEGIN
  ...
  RETURN 5;
  ...
  END;
```

In this routine, the apparent return value 5 is discarded, because the routine has the NOVALUE attribute.

However, notice the following case:

```
ROUTINE S(B) =
  BEGIN
  ...
  RETURN;
  ...
  END;
```

Here an informational message is issued indicating that a value expression is missing in a context that implies a value. (The compiler assumes a value of zero for missing expressions.)

5.2.3 Semicolons and Values of Blocks

It is common to think of the semicolon as a terminator for an expression, but this is not always true and can lead to errors. For example:

```
IF .A
THEN
  X=.Y;
ELSE
  X=-5;
```

Here the first semicolon terminates the initial IF-THEN and the subsequent ELSE is in error.

A more subtle error is to place a semicolon after the last expression of a block when that expression is supposed to be the value of the block. (This is very similar to Section 5.2.2.)

5.2.4 Complex Expressions Using AND, OR, and NOT

When you are writing complex tests involving the AND, OR, and NOT operators, it is easy to confuse the relative precedence of the operators. Use parentheses to make your intent explicit to the compiler, to other readers, and to yourself. For example, the following construction is difficult to read and may not reflect the intention:

```
IF .X EQL 0 AND .Y OR NOT .J THEN...
```

The following clearly reflects the intent:

```
IF ((.X EQL 0) AND .Y) OR (NOT .J) THEN...
```

5.2.5 Computed Routine Calls

When computing the address of a routine to be called, enclose the expression that computes the address in parentheses followed by the parameters. For example:

```
BEGIN
EXTERNAL ROUTINE
  R;
LOCAL
  L;
L = R;
(.L)(0)
END
```

This code calls the routine at address R with a parameter of zero. However, notice the following call:

```
.L(0)
```

This expression calls the routine at address L (most likely an address on the stack) and uses the returned value as the operand of the fetch. Because there is no code at address L, an illegal instruction exception is likely at execution.

An alternative is to use a general routine call. Assuming the desired linkage is the default calling convention, you could write the computed call as follows:

```
BLISS(.L,0)
```

5.2.6 Signed and Unsigned Fields

Be careful when using signed and unsigned fields that are smaller than a fullword. Consistent use of the sign extension rules and signed versus unsigned operations is important. For example:

```
BEGIN
FIELD LOW9 = [0,0,9,0];
OWN
   X : BLOCK[1] FIELD(LOW9);
THEN ...
...
END
```

Here the expression .X[LOW9] EQL–5 is always false because the (unsigned) value fetched from X is necessarily in the range 0 to 511.

5.2.7 Complex Macros

The BLISS macro facility has many capabilities, but also has some very subtle properties. Most problems arise when features are used that on compilation produce unwarranted side effects, such as the following:

- Macro expansions that produce declarations of any kind, particularly other macro declarations
- Use of compile-time names to control macro expansions using %ASSIGN
- Use of %QUOTE, %UNQUOTE, and %EXPAND

Be careful when using these features; indeed, you may not really need them. (Refer to Section 5.6 for examples of advanced macro usage.)

5.2.8 Missing Code

Many coding errors tend to result in code that can be optimized. If you discover that some of your program seems to be missing from the compiled code, you may possibly have made a coding error. Check the compiled code carefully to ensure that the code is missing from the source, rather than removed by optimization.

For example, consider the following:

```
BEGIN
OWN
 X: BYTE;
...
IF .X EQL -5 THEN X = 0;
...
END
```

In this example, the value of the test expression, .X EQL–5, is always false (refer to Section 5.2.6.) As a result, the compiler produces no code for the test expression or the alternative (X=0), which can never be executed.

Consequently, the entire IF expression disappears from the compiled code. The problem is not erroneous compiler optimization but a missing SIGNED attribute in the declaration of X.

A similar error occurs if the fact is overlooked that TRUE_/FALSE is based on the value of the low bit of an expression. For example:

```
IF .X and 2
THEN
  Y=0
ELSE
  Y=1
```

This code will always assign the value 1 to Y, because the low bit of .X AND 2 must always be zero or false.

5.2.9 Using the Built-In PC

The *VAX Architecture Handbook* asserts that the PC cannot be used in REGISTER or REGISTER indirect addressing modes. The BLISS–32 compiler generates an internal compiler error if the following kinds of expressions are encountered:

```
BUILTIN PC;
OWN
    X;
X = .PC;    ! produces a MOVL PC,X which is unpredictable
X = ..PC;   ! produces a MOVL (PC),X which is unpredictable
```

If it is necessary for you to obtain an approximate value for the PC, the technique to use is as follows:

```
BUILTIN PC;
EXTERNAL ZERO:WEAK;   ! default to zero if no corresponding
                      ! GLOBAL BIND ZERO = 0 appears
    X = .PC + ZERO;
```

This will produce the following:

```
MOVAB ZERO(PC),X
```

This instruction is defined in the *VAX Architecture Handbook*.

5.2.10 Indexed Loop Coding Error

A common coding error occurs in the use of unsigned indexed loop expressions of the following form:

```
DECRU I FROM HIGH TO 0 DO
```

This results in an infinite loop because the programmer is inadvertently attempting to decrement through zero. The rules of unsigned arithmetic state that zero is the smallest integer; thus, .I is never less than zero and the loop cannot terminate.

The proper interpretation of these expressions is as follows:

```
BEGIN
  LOCAL I;
  I = HIGH;
  IF .I GTRU 0 DO ( I = .I - 1) UNTIL .I LSSU 0;
END;
```

Thus, when .I is zero and is decremented to –1, it becomes the largest unsigned number. The expression .I LSSU 0 is always false because –1 LSSU 0 is always false. Therefore, the unsigned indexed loop expression must be coded as either one of the following:

```
DECRU I FROM HIGH + 1 TO 1 DO
```

or

```
DECRU I FROM HIGH TO 0 DO
  BEGIN
  ...
    IF .I EQL 0 THEN EXITLOOP
  END;
```

The semantics of the DECRA expression are the same, except GTRU and LSSU would be GTRA and LSSA.

5.3 Linker Error Messages

The TRUNC or TRUNCDAT error message is typically caused by a module compiled with the following default addressing mode being linked in an image that is larger than 32K:

```
ADDRESSING_MODE(WORD_RELATIVE)
```

The problem can be resolved with one of the following procedures:

- Edit the source module to use ADDRESSING_MODE(LONG_RELATIVE) and recompile.
- Rearrange the placement of the object modules so that EXTERNAL references or references between program sections are within 32KB of their target.

5.4 Obscure Error Messages

Obscure error messages that appear after a program has been run are typically caused by a program whose main routine fails to return a valid VMS completion code.

5.5 Position-Independent Code Generation

The BLISS–32 compiler always generates position-independent code (PIC) for expressions that involve relocatable quantities. This may cause surprisingly complex code to be generated. However, it is the programmer's responsibility to ensure that PIC data has been generated when necessary.

5.6 Advanced Use of BLISS Macros

This section provides some examples of the advanced use of BLISS macros. In particular, the examples demonstrate the use of the following:

- Conditional compilations
- Iterative macros
- Lexical functions (such as %QUOTE and %EXPAND)

5.6.1 Advantageous Use of Machine Dependencies

The following examples show how machine-independent constructs can be modified to take advantage of machine dependencies.

Assume a high-level construct is needed to move a fullword sequence from a source to a destination; the simplest transportable implementation of this might be as follows:

```
MACRO
  MOVECORE( SRC, DST, LENTH ) =
    BEGIN
    BIND
      $S=(SRC) : VECTOR,
      $D=(DST) : VECTOR;

    INCR I FROM 0 TO (LENTH)-1 DO $D[.I] = .$S[.I]
    END %;
```

Notice that the BIND of SRC and DST guards against their producing any extraneous side-effects. For example, if the assignment-expression in the INCR loop used a general structure reference of the following form, and the DST or SRC expressions were routine calls, a call would be executed every time a pass was made through the loop:

```
VECTOR[ DST, .I ] = .VECTOR[ SRC, .I ];
```

Thus, the BIND to $D and $S ensures that this side effect occurs only once.

5.6.2 Using String Instructions

The macro shown in the previous example is, however, inefficient for use in the moving of large blocks of memory due to excessive execution time and instruction size; more efficient coding would take advantage of string instructions supported by the VAX and DECsystem 10/20 hardware.

As an example, the transportable CH$MOVE function can be used, with appropriate adjustments, to deal with 8-bit bytes on the VAX and 36-bit bytes on the 10/20 as follows:

```
MACRO
  WORD_PTR(A) = CH$PTR( A %BLISS36(, 0, 36) ) %,

      MOVECORE(SRC,DST,LENTH) = CH$MOVE(
                                      (LENTH)*%UPVAL,
                                      WORD_PTR(SRC),
                                      WORD_PTR(DST) ) %;
```

This method produces fairly efficient code; but for DECsystem 10/20 an even more efficient implementation is possible using the BLT (Block Transfer) instruction as follows:

```
MACRO
  MOVECORE(SRC,DST,LENTH)=
    %IF %BLISS(BLISS36)
    %THEN
      BEGIN
      BIND
        $D = (DST);
      BUILTIN MACHOP;
      LITERAL BLT=%O'251';    ! BLT opcode
      REGISTER
        RQQQ,
        SQQQ=1;               ! Must not be AC0

      RQQQ<18,18> = (SRC);    ! Source-ptr in LH
      RQQQ<0,18> = ($D);      ! Destination in RH

      %IF %LTCE( LENTH )      ! Effective address
      %THEN                   ! tells where to stop
        SQQQ = .RQQQ<0,18>;
        MACHOP( BLT, RQQQ, (LENTH)-1, SQQQ )
      %ELSE
        SQQQ = ($D) + (LENTH);
        MACHOP(BLT, RQQQ, -1, SQQQ)
      %FI
      END
    %ELSE
      CH$MOVE((LENTH)*%UPVAL, SRC, DST)
    %FI %,
```

The BLISS–36 version of MOVECORE is now heavily conditionalized to generate the best possible code when the length of the memory block is known at compile or link time.

For example, the %LTCE lexical function is used to determine if advantage can be taken of the PDP-10 effective-address calculation to completely evaluate the expression:

```
.SQQQ + (LENTH) - 1
```

However, if the expression is not a link-time constant, it is necessary to compute the ending address by a combination of run-time addition and effective-address computation.

Further, it should be noted that there is some risk involved with this implementation, in that the BLT always moves at least one word. Thus, if the LENTH parameter is zero, the BLT method will incorrectly move one word to the destination.

5.6.3 Dealing with Enumeration Types

One of the more powerful features of languages such as Pascal is the ability to define a finite set of elements. This is known as an "enumerated type", because you define the set by exhaustively naming each possible element in the set.

While BLISS is unaware of such types, it is possible to provide an adequate simulation.

5.6.3.1 The SET Data Type

Sets are implemented in BLISS through a system of macros. An element of the set is given a small integer value between zero and %BPVAL–1. Thus, sets are limited (in this implementation) to contain no more than %BPVAL elements in their domain.

You store a subset by setting bits in a fullword. For reasons to be discussed later, the bits in the fullword are numbered in reverse order from the normal BLISS conventions. Thus for the VAX a set looks as follows:

```
31                                      0
+--+--+--+--+--+--+--+--+--------+--+--+--+
| 0| 1| 2| 3| 4| 5| 6| 7|........|29|30|31|
+--+--+--+--+--+--+--+--+--------+--+--+--+
```

The digits inside the box refer to element-numbers, while the digits outside the box refer to the most-significant (31) and least-significant bits (0) of a VAX longword.

The following table shows some simple sets and the elements they contain:

Set(Hexadecimal)	Contents
80000000	0
80000001	0,31
00000002	30
FF000000	0,1,2,3,4,5,6,7

5.6.3.2 Creating a Set

The following macro defines an enumeration type.

```
MACRO
  ENUMERATION(NAME)=
    !+
    ! Creates a PASCAL-like ENUMERATION type. The parameter
    ! NAME will be defined as a macro expanding to the
    ! comma-list of the %REMAINING parameters.
    !-
    COMPILTIME
      NAME = 0;

    LITERAL
      ENUM_ (NAME,%REMAINING);

    UNDECLARE NAME;
    MACRO NAME = %QUOTE %EXPAND %REMAINING %QUOTE %
    %,

  ENUM_(V)[N] = N = V %ASSIGN(V,V+1) %;
```

Note that the ENUMERATION macro is particularly interesting due to the use of the %QUOTE lexical function. As an example, consider the use of the ENUMERATION macro to define the type TREES:

```
ENUMERATION( TREES, OAK, MAPLE, PINE, ELM );
```

The intended result is that the names OAK, MAPLE, PINE, and ELM should be defined as follows:

```
LITERAL
  OAK  = 0,
  MAPLE= 1,
  PINE = 2,
  ELM  = 3;
```

And the name TREES should be defined by the following macro:

```
MACRO TREES = OAK, MAPLE, PINE, ELM %;
```

You must use %QUOTE with %EXPAND to keep it from occurring until you expand the ENUMERATION macro. At this time, the %REMAINING will be bound to the list of names: OAK, MAPLE, PINE, ELM.

Because the macro NAME is inside another macro, you are scanning its body at macro-quote level when you expand the ENUMERATION macro. Thus, suppose the body of the ENUMERATION macro was defined as follows:

```
...
MACRO NAME = %REMAINING %QUOTE %;
...
```

The result of expanding ENUMERATION would be as follows:

```
MACRO TREES = %REMAINING %;
```

Because TREES is defined as not having a macro-parameter list, the value of the %REMAINING would always be empty. Therefore, you must force the expansion of %REMAINING when the ENUMERATION macro is expanded, not when the TREES macro is expanded.

Suppose you have defined the ENUMERATION macro as follows:

```
...
MACRO NAME = %EXPAND %REMAINING %QUOTE %;
...
```

This would cause %REMAINING to be expanded too soon (when the ENUMERATION macro is declared), and again %REMAINING would be an empty lexeme sequence.

The macro, which expands to the entire domain of the enumeration type, can be used as follows:

1. To iterate over a set, write the following:

```
INCR SPECIES FROM MIN(TREES) TO MAX(TREES) do ...
```

2. Using the CASE control expression, write the following:

```
CASE .WOOD FROM MIN(TREES) TO MAX(TREES) OF
  SET
  [OAK]:        ...
  [MAPLE]:      ...
  [PINE,ELM]:   ...
  [OUTRANGE]:   ...
  TES;
```

3. A successor function can be defined as follows:

```
MACRO SUCCESSOR(T) = ( (T)+1 ) MOD MAX(TREES) %;
```

The MOD is used to cause SUCCESSOR(MAX(TREES)) to be the same as MIN(TREES).

A more general implementation would pass the SET as a parameter, as follows:

```
MACRO SUCCESSOR(T) = ((T)+1) MOD MAX(%REMAINING) %;
```

This would be invoked as SUCCESSOR(MAPLE, TREES) to return the value 2 (the literal associated with PINE).

5.6.3.3 Placing Elements in Sets

Given elements of an enumeration type, you can produce a subset containing those elements. The SUBSET macro shows another useful example of an iterative macro. Its purpose is to OR together the single sets produced by BITS. Notice how the sequence “0 OR BITS(. . .)” is used to force the default separator to be OR. BITS is notable for the inclusion of defensive code to detect attempts to produce sets that exceed the implementation limits.

```
MACRO
  BITS[A]=
    %IF %CTCE(A)
    %THEN
      %IF A GTRU %BPVAL-1
      %THEN
        %WARN('Value (', %NUMBER(A),
           ') in BITS Mask exceeds %BPVAL-1')
      %FI
    %FI
    ( 1^( (%BPVAL-1)-(A) ) ) %,

  SUBSET[] = ( 0 OR BITS(%REMAINING) ) %;
```

Typical use of the SUBSET macro would be to initialize a set or to test for intersections, as follows:

```
OWN FOREST : INITIAL( SUBSET<OAK, MAPLE> );

...
IF (.FOREST AND SUBSET<PINE,ELM>) EQL 0
THEN
  ! No trees in common between two sets
```

5.6.3.4 Membership in a Set

The ONEOF macro efficiently determines if an element is a member of a given subset by doing a left-shift and examining the sign bit of the result value. For this reason, sets are stored in reverse-numbered bit fields.

This example also deals with machine dependencies, as the BLISS–36 arithmetic shift operator (^) must choose either an ASH or a LSH instruction. An ASH instruction leaves the sign bit unchanged. This is the behavior wanted when right-shifting any value, but when left-shifting, the BLISS semantics demand that a LSH be generated. If the shift count is unknown at compile time, BLISS–36 must generate a run-time test and conditionally execute either the ASH or the LSH. Because the shifts will always be to the left, the generated code is optimized by forcing a LSH to be emitted as shown below.

```
MACRO
  ONEOF(ELEMENT,SUBSET)=
    %IF %BLISS(BLISS36)
    %THEN
      BEGIN
      BUILTIN LSH;
      LSH(SUBSET, ELEMENT) LSS 0
      END
    %ELSE
      (((SUBSET) ^ (ELEMENT)) LSS 0)
    %FI %;
```

For example, on the VAX, consider the following:

```
LOCAL TREE : INITIAL( ELM );   ! An element

...
IF ONEOF( .TREE, SUBSET<ELM, PINE> )
THEN
  ...
```

This code fragment would expand to the following:

```
IF ( (1^(31-3)) OR (1^(31-2)) )^(.TREE) LSS 0
THEN
```

This is equivalent to the following:

```
IF (%X'30000000' ^ (.TREE)) LSS 0
THEN
  ...
```

Assuming that TREE still contains its initial value of 3 (for ELM), this is equivalent to the following:

```
IF (%X'80000000') LSS 0
THEN
  ...
```

This evaluates to TRUE and indicates that ELM is a member of the subset {ELM, PINE}.

6 Transportability Guidelines

This chapter provides guidelines for writing transportable programs and demonstrates why the writing of transportable code is easier if it is considered at the beginning of a project. Also described are ways in which a program can become untransportable, and techniques whereby such occurrences can be avoided.

Following an introduction to the concepts of transportability, the guidelines are presented in three sections: Section 6.2 describes high-level approaches to writing transportable software in BLISS. Section 6.3 describes various BLISS features that can be used in solving transportability problems. Section 6.4 analyzes various transportability problems and provides suggested solutions.

The dialects referenced in this chapter are the languages defined by the following BLISS compilers:

BLISS–16 Version 4.3
BLISS–32 Version 4.3
BLISS–36 Version 4.0

BLISS–36 and BLISS–16 are generic names for languages that have BLISS compilers that generate code for DECsystem-10/20s and PDP–11s, respectively.

6.1 Introduction

A transportable BLISS program is one that can be compiled to execute on at least two, and preferably all, of the three major architectures: PDP-10, PDP–11, and VAX. Various solutions to the problem of transportability exist, each requiring different levels of effort. Various kinds of solutions are recommended. For example, in some cases, program text should be rewritten. However, large portions of programs can be written in such a way that they will require no modification and yet be functionally equivalent in differing architectures. The levels of solutions in order of decreasing desirability are as follows:

- No change to program text is needed—the program is completely transportable.
- Parameterization solves the transportability problem—the program makes use of some features that have an analog on all the other architectures.
- Parallel definitions are required—either the program makes use of features of an architecture that do not have analogs across all other architectures, or different, separately transportable aspects of the program interact in nontransportable ways.

The goal is to make transportability as simple as possible, which means that the effort needed in transporting programs should be minimized. Central to the ideas presented here is the notion that transportability is more easily accomplished if considered from the beginning. Transporting programs after they are running becomes a much more complex task.

It is advantageous to run parallel compilations frequently. It is fortunate, therefore, that with the right tools and techniques transportability is not difficult to achieve. The first transportable program is the hardest. Before undertaking a large programming project, you may find it useful to write and transport a less ambitious program.

These guidelines are the result of a concentrated study of the problems associated with transportability. No claim is made that these guidelines are complete; however, an attempt has been made to identify less obvious problem areas and provide solutions.

6.2 General Strategies

Two global strategies are important to the writing of transportable BLISS programs: isolation and simplicity.

6.2.1 Isolation

Remember the following rule when you are designing or coding a program that is to be transported: If a construct is not transportable, isolate it.

You will probably encounter situations for which it is desirable to use machine-specific constructs in your BLISS program. In such cases, isolating the constructs will facilitate any future movement of the program to a different machine. Generally, only a small percentage of the program or system will be sensitive to the machine on which it is running. By isolating those sections, you can mainly confine the effort involved in transporting the program to these easily identifiable, machine-specific sections. Specifically, follow these rules:

- If machine-specific data are to be allocated, place the allocation in a separate module or in a REQUIRE file.
- If machine-specific data are to be accessed, place the accessing code in a routine or a macro and then place the routine or macro in a separate module or a REQUIRE file.
- If a machine-specific function or instruction is to be used, isolate it by placing it in a REQUIRE file.
- If it is impossible or impractical to isolate this part of your program from its module, comment it heavily. Make it obvious to the reader that the code is nontransportable.

These rules are applicable in the local context of a routine or module. In a larger or more global context (for instance, in the design of an entire system), isolation is implemented by the technique of modularization. By separating those parts of the system that are machine or operating-system dependent from the rest of the system, the task of transporting the entire system is simplified. It becomes a matter of rewriting a small section of the total system. The major portion of the code (if written in a transportable manner) should be easy to move to a new machine with a minimum of rewriting. The BLISS language facilitates both the design and programming of programs and systems in a modular fashion. You should use this feature to advantage when writing a transportable system.

6.2.2 Simplicity

A basic concept in writing transportable BLISS software is simplicity in the use of the language. BLISS was originally developed for the implementation of systems software. As a result of this, BLISS is one of few high-level programming languages that allow ready access to the machine on which the program will be running. You are allowed to have complete control over the allocation of data, for example. Unfortunately, the same language features that allow access to underlying features of the hardware are the very features that lead to nontransportable code. In a system intended to be transportable, these features should be used only where necessary to meet a specific functional, performance, or economy objective.

It is often the case that BLISS language features that make a program nontransportable also make the program more complex. Reducing the complexity of data allocation, for example, results in a transportable subset of the BLISS language. This reduction of complexity is one of the basic themes that runs through these guidelines. In effect, the writing of transportable programs is a simpler task because the number of options available has been reduced. Simplicity in the writing effort is one of the reasons for the development of higher-level languages like BLISS. The use of the defaults in BLISS will result in programs that are much more easily transported.

6.3 Tools

This section on tools presents various language features that provide a means for writing transportable programs. These features are either normal features of BLISS or have been designed for transportability or software engineering uses. The tools described here will be used throughout Section 6.4 as well.

6.3.1 Literals

Literals provide a means for associating a name with a compile-time constant expression. This section considers some predeclared or built-in literals that aid in writing transportable programs. In addition, it discusses restrictions on user-defined literals.

6.3.1.1 Predeclared Literals

One of the key techniques in writing transportable programs is parameterization. Literals are a primary parameterization tool. The BLISS language has a set of predeclared, machine-specific literals that can be useful. These literals parameterize certain architectural values of the three machines. The values of the literals are dependent on the machine for which the program is currently being compiled. Here are their names and values:

Literal Name	Description	DEC 10/20	VAX	PDP–11
%BPUNIT	Bits per addressable unit	36	8	8
%BPADDR	Bits per address value	18	32	16
%BPVAL	Bits per BLISS value	36	32	16
%UPVAL	Units per BLISS value	1	4	2

The names beginning with a percent sign (%) are literal names that can be used without declaration. These literal names are used throughout this chapter.

Bits per value is the maximum number of bits in a BLISS value. Bits per unit is the number of bits in the smallest unit of storage that can have an address. Bits per address refers to the maximum number of bits an address value can have. Units per value is the quotient %BPVAL/%BPUNIT. It is the maximum number of addressable units associated with a value.

You can derive other useful values from these built-in literals. For example:

```
LITERAL
    HALF_VALUE = %BPVAL / 2;
```

This defines the number of bits in a half word (half a longword on the VAX).

6.3.1.2 User-Defined Literals

Strictly speaking, a literal is not a self-defining term. You arrive at the value and restrictions associated with a literal by assigning certain semantics to its source program representation. It is convenient to define the value of a literal as a function of the characteristics of a particular architecture, which means that there are certain architectural dependencies inherent in the use of literals. Because the size of a BLISS value determines the value and the representation of a literal, there are some transportability considerations.

BLISS value (machine word) sizes are different on each of the three machines. On VAX, the size is 32 bits; on the DECsystem 10/20 systems, it is 36; and on the PDP–11, the value is 16.

There are two types of BLISS literals: numeric literals and string literals. The values of numeric literals are constrained by the machine word size. The ranges of values for a signed number (i) are as follows:

VAX	–(2**31)	≤ i ≤ (2**31)-1
DECsystem-10/20	–(2**35)	≤ i ≤ (2**35)-1
PDP–11	–(2**15)	≤ i ≤ (2**15)-1
All	–(2**(%BPVAL–1))	≤ i ≤ (2**(%BPVAL-1))-1

A numeric literal (%C'single-character') has been implemented. Its stored value is the ASCII code corresponding to the character in quotes. It is right-justified in a BLISS value (word or longword). A more thorough discussion of its usage can be found in Section 6.4.3. There are two ways of using string literals: as integer values and as character strings. When string literals are used as values, they are not transportable. This arises out of the representational differences and from differing word sizes. The following table illustrates these potential differences for an %ASCII type string literal:

	VAX	DECsystem-10/20	PDP–11
Maximum number of characters	4	5	2
Character placement	right to left	left to right	right to left

This type of string-literal usage and also its use as a character string are discussed in Section 6.4.3.

6.3.2 Macros and Conditional Compilation

BLISS macros can be an essential tool in the development of transportable programs. Because they evaluate (expand) during compilation, it is possible to use macros to tailor a program to a specific machine.

A good example can be found in the Section 6.4.5.1, where two macros are developed whose functions are completely transportable. The macros can determine the number of addressable units needed for a vector of elements, where the element size is specified in terms of bits. There are also predefined machine conditionalization macros available. Depending on which compiler is being run, you can use these macros to select declarations or expressions for compilation. There are three sets of definitions, each containing three macro definitions.

The definitions for the BLISS–32 set are as follows:

```
MACRO
    %BLISS16[] = % ,
    %BLISS36[] = % ,
    %BLISS32[] = %REMAINING % ;
```

There are analogous definitions for the other machines. The net effect is that in the BLISS–32 compiler, the arguments to %BLISS16 and %BLISS36 will disappear, while arguments to %BLISS32 will be replaced by the text given in the parameter list.

A very explicit way of tailoring a program to a specific architecture uses the %BLISS lexical function in conjunction with the conditional compilation facility in BLISS. The %BLISS lexical function takes either BLISS36, BLISS32 or BLISS16 as a parameter, and returns 1 if the parameter corresponds to the compiler currently executing, and 0 otherwise. In the following example, INSQUE is an executable function in BLISS–32, but is a routine for BLISS–36:

```
%IF %BLISS(BLISS32)
%THEN
    BUILTIN
        INSQUE;
%ELSE
    %IF %BLISS(BLISS36)
    %THEN
    FORWARD ROUTINE
        INSQUE;
    %FI
%FI
```

6.3.3 Module Switches

A module switch and a corresponding special switch are provided to aid in the writing of transportable programs. This switch, LANGUAGE, is provided for two reasons:

- To indicate the intended transportability goals of a module
- To provide diagnostic checking of the use of certain language features

Using this switch, you can therefore indicate the target architectures (environments) for which a program is intended.

Transportability checking consists of the compiler determining whether, in the module being compiled, certain language features appear that fall into either of two categories:

- Features that are not commonly supported across the intended target environments
- Features that most often prove to be troublesome in transporting a program from any one environment to another

The syntax is as follows:

```
LANGUAGE (language-name ,...)
```

where language-name is any combination of BLISS36, BLISS16, or BLISS32.

Two other forms are possible:

```
LANGUAGE(COMMON)
LANGUAGE()
```

If no LANGUAGE switch is specified, the default is the single language name corresponding to the compiler used for the compilation, and no transportability checking is performed. If more than one language-name is specified, the compiler assumes that the program is intended to run under each corresponding architecture.

If no language name is specified, no transportability checking will be performed. A specification of COMMON is the equivalent of the specification of all three.

Each compiler will give a warning diagnostic if its own language-name is not included in the list of language-names.

Within the scope of a language switch, each compiler gives a warning diagnostic for any nontransportable or problematic language construct relative to the specified set of languages. This chapter discusses most of the constructs that will be checked.

The precise set of language constructs that are subject to transportability checking is specified in the *BLISS Language Reference Manual*.

Here is an example of how the LANGUAGE switch can be used:

```
MODULE FOO(...,LANGUAGE(COMMON),...) =
BEGIN
!+
! Full Transportability Checking is in effect.
!--

               ...
               ...
               ...

BEGIN
!+
! BLISS36 no longer in effect: BLISS--16/32 Subset
! checking to be performed in this block.
!--
SWITCHES
        LANGUAGE(BLISS16, BLISS32);
                ...
                ...
                ...
```

```
        Within this block (that is, within the scope
        of the SWITCHES declaration) a relaxed form of
        full transportability checking is performed.
        (This takes advantage of the greater degree
        of commonality that exists between BLISS--16
        and BLISS--32 target architectures.)

        The compilation of this section of code by a
        BLISS--36 compiler will result in a disgnostic
        warning.

                ...
                ...
                ...
END;
!+
!  Full transportability checking is restored.
!--
```

6.3.4 Reserved Names

The following is a list of BLISS reserved names that cannot be declared by the user. Although the same names are reserved in all three BLISS dialects, some of them do not have a predefined meaning in each dialect. For example, LONG is an allocation-unit keyword in BLISS–32 and is a reserved but otherwise unsupported name in BLISS–16 and BLISS–36 (due to basic architectural differences in the target systems). Any attempted use of this name in the latter two dialects will result in a compiler diagnostic. As another example, the name IOPAGE has no defined meaning in any BLISS dialect but is reserved for possible future use in all dialects. The reserved names that are not supported in some or all dialects are marked with an asterisk. See the *BLISS Language Reference Manual* for a more complete description.

*ADDRESSING_MODE	GTRU	PLIT
*ALIGN	IF	PRESET
ALWAYS	INCR	PSECT
AND	INCRA	*RECORD
BEGIN	INCRU	REF
BIND	INITIAL	REGISTER
*BIT	INRANGE	REP
BUILTIN	*IOPAGE	REQUIRE
BY	KEYWORDMACRO	RETURN
*BYTE	LABEL	ROUTINE
CASE	LEAVE	SELECT
CODECOMMENT	LEQ	SELECTA
COMPILETIME	LEQA	SELECTONE
DECR	LEQU	SELECTONEA
DECRA	LIBRARY	SELECTONEU
DECRU	LINKAGE	SELECTU
DO	LITERAL	SET
ELSE	LOCAL	*SHOW
ELUDOM	*LONG	*SIGNED

ENABLE	LSS	STACKLOCAL
END	LSSA	STRUCTURE
EQL	LSSU	SWITCHES
EQLA	MACRO	TES
EQLU	MAP	THEN
EQV	MOD	TO
EXITLOOP	MODULE	UNDECLARE
EXTERNAL	NEQ	*UNSIGNED
FIELD	NEQA	UNTIL
FORWARD	NEQU	UPLIT
FROM	NOT	VOLATILE
GEQ	NOVALUE	*WEAK
GEQA	OF	WHILE
GEQU	OR	WITH
GLOBAL	OTHERWISE	*WORD
GTR	OUTRANGE	XOR
GTRA	OWN	

6.3.5 REQUIRE and Library Files

REQUIRE files are a way of gathering machine-specific declarations and expressions together in one place. Library files are a form of precompiled REQUIRE files.

In many cases, it will be either impossible or unnecessary to write a particular BLISS construct (for example, routines and data declarations) in a transportable manner. Developing parallel REQUIRE files, one for each machine, can often provide a solution to transporting these constructs.

For example, if a certain set of routines is very machine specific, then the solution may be to write two or three functionally equivalent routines, one for each machine type, and segregate them each in their own REQUIRE file.

Each BLISS compiler has a predefined search rule for REQUIRE file names based on their file types. Each compiler will search first for a file with a specific file type, then it will search for a file with the file type .BLI.

The search rules for each compiler are as follows:

Compiler	1ST	2ND	3RD	4TH
BLISS--36	.R36	.REQ	.B36	.BLI
BLISS--16	.R16	.REQ	.B16	.BLI
BLISS--32	.R32	.REQ	.B32	.BLI

Hence, the following REQUIRE declaration searches for IOPACK.R36, IOPACK.R16, or IOPACK.R32, depending on which compiler is run.

```
REQUIRE
  'IOPACK';            ! I/O Package
```

If these searches fail, it then looks for IOPACK.REQ, and so on.

Inherent in these search rules is a naming convention for REQUIRE files. If the file is transportable, give it the file type .REQ or .BLI. If it is specific to a particular dialect, give it the corresponding file type (for example, .R36 or .B36).

Each BLISS compiler, by the use of the /LIBRARY switch, is capable of precompiling files containing declarations. However, not all declarations can be processed in a library run; those that are allowed are described in the *BLISS Language Reference Manual*. The output of a library run is called a library file; library files are processed by a compiler when it encounters a LIBRARY declaration. For example:

```
LIBRARY
    'IOPACK';
```

Each compiler checks to see that the library file it is using was produced by itself in a previous run. Thus, to build a transportable library from a single transportable source, you must build unique LIBRARY files for each architecture of interest, using the appropriate compilers of interest.

For example, let us assume that the file SYSDCL.BLI contains a set of transportable declarations common to an application that is to run on a DECSYSTEM-20 and a VAX. To precompile it requires that it be run on the BLISS–32 compiler with the /LIBRARY switch and the BLISS–36 compiler using the /LIBRARY switch. The object file produced by the compiler is the library file, and if no extension is given for it in the command line, a default extension is used (for example, .L32 and .L36 respectively).

6.3.6 Routines

The key to transportability is the ability to identify an abstraction that can exist in several environments. You do this by naming the abstraction and describing its external characteristics in a way that permits implementation in any of the environments. The abstraction can then be implemented separately in each environment. The closed subroutine has long been regarded as the principal abstraction mechanism in programming languages. With BLISS, other abstraction mechanisms are also available, such as structures, macros, literals, REQUIRE files, and so on, but the routine can still be easily used as a transportability abstraction mechanism.

For instance, when designing a system of transportable modules which uses the concept of floating-point numbers and associated operations, you will need to perform floating-point arithmetic. The question naturally arises as to the environment in which the arithmetic should be done. If the floating-point arithmetic resides entirely in a well-defined set of routines, and no knowledge of the various representations of floating-point numbers is used except through these well-defined interface routines, then it becomes possible to perform "cross-arithmetic", which is important, for instance, when writing cross-compilers. Even if the ability to perform cross-arithmetic is not desired, isolating floating-point operations in routines may be a good idea because these routines can then be reused more easily in another project. A little thought will indicate that the floating-point routines themselves have to be transportable if they are going to perform cross-arithmetic (because the system under construction is transportable), but need not be transportable if cross-arithmetic is not a goal.

The principal objection to using routines as an abstraction mechanism is that the cost of calling a procedure is significant, and that cost is strictly program overhead. Composing routine abstractions produces serious performance degradations. For this reason, you should not attempt to use a routine as a transportability mechanism when a small amount of effort is sufficient for the writing of a single transportable module.

6.4 Techniques for Writing Transportable Programs

This section on techniques shows you how to write transportable programs. The section is organized in dictionary form by BLISS construct or concept. Each subsection contains the following:

- A discussion of the construct or concept
- Transportability problems that its use may engender
- Specific guidelines and restrictions on the use of the construct or concept
- Examples—both transportable and nontransportable

In all cases, the examples attempt to use the tools described in the previous section.

6.4.1 Allocation of Data

This section deals with the allocation of data in a BLISS program. In this section, character sequence (string) data or the formation of address data is not discussed. These types of data are discussed in Section 6.4.2 and Section 6.4.3. This section describes the allocation of scalar data (for example, counters, integers, pointers, addresses, and so on.) A presentation of more complex forms of data can be found in Sections 6.4.4 and 6.4.5.

The first discussion concerns transportability problems encountered due to differing machine architectures. This is followed by a presentation on the BLISS allocation-unit attribute, and then a discussion of other BLISS data attributes you must consider when writing transportable programs.

6.4.1.1 Problem Origin

The allocation of data (through OWN, LOCAL, GLOBAL, and other declarations) tends to be one of the most sensitive areas of a BLISS program in terms of transportability. This problem of transporting data arises chiefly from two sources:

- The machine architectures
- The flexibility of the BLISS language

In considering writing a BLISS program for transporting to another machine, you will be confronted with the problem of allocating data on at least two different architectures.

Although differing word sizes have already been discussed, there are further differences to be considered. On VAX architecture, for example, data may be typically fetched in longwords (32 bits), in words (16 bits), and in bytes (8 bits); while on the PDP–11, both words and bytes may be fetched; and on the DECsystem-10/20 only 18-bit halfwords and 36-bit words can be fetched without a byte pointer.

What decisions must you make in the transportable allocation of data? Need you be concerned with how many bits are going to be allocated?

These questions can be further complicated by the other chief source of data transportability problems—namely, the BLISS language itself.

BLISS is unlike many other higher-level languages in that it allows ready access to machine-specific control, particularly in storage allocation. This helps you write efficient machine-specific software. You have much more control over exactly how many bits of data you will use. This feature of BLISS can, however,

complicate the decisions you must make regarding transportability. For example, do you allocate scalars by bytes, words, or longwords?

6.4.1.2 Transportable Declarations

Consider the following simple example of a data declaration in BLISS–32:

```
LOCAL
    PAGE_COUNTER: BYTE;       ! Page counter
```

One byte (eight bits) is allocated for a variable named PAGE_COUNTER. No matter what the intention was in requesting only one byte of storage, this declaration is nontransportable. The concept of BYTE (in this context) does not exist on the DECsystem-10/20. In fact, in BLISS–36 the use of the word BYTE results in an error message. Moreover, because the storage is allocated on the stack or in a register, there is even less motivation to make it a byte due to the frequent use of these locations.

Suppose this declaration had been originally written as follows:

```
LOCAL
    PAGE_COUNTER;       ! Page counter
```

This could have been transported to any of the three machines. The functionality (in this case, storing the number of pages) has not been lost. By not specifying any allocation unit in the LOCAL declaration, the BLISS compiler is allowed to allocate storage by default. In all BLISS dialects the default size for an allocation unit consists of %BPVAL bits. Thus, the first transportable guideline is as follows:

- Do not use the allocation-unit attribute in a scalar data declaration.

In the case of scalar data, using the default allocation unit sometimes results in the allocation of more storage than is strictly necessary. This gain in program data size (which is small in most instances) must be weighed against the decrease in fetching time for particular scalar values, and the knowledge that because of the default alignment rules no storage savings may, in fact, be realized.

In the BLISS language, the default size of %BPVAL bits was chosen (among other reasons) because this is the largest and most efficiently accessed unit of data for a particular machine. In other words, saving bits does not necessarily mean a more efficient program has been written.

There are other attributes besides the allocation unit that, if used, may present transportability problems. In particular, when allocating data, adhere to the following guidelines:

- Do not use the following attributes:

  SIGNED
  UNSIGNED
  ALIGN
  WEAK

In other words, before writing a declaration, consider whether you need to specify any data attributes other than structure attributes.

The extension attribute (SIGNED and UNSIGNED) specifies whether the sign bit is to be extended in a fetch of a scalar (or equivalently, whether or not the leftmost bit is to be interpreted as a sign bit). In any case, no sign extension can be performed if the allocation unit is not specified.

The alignment attribute (ALIGN) specifies the kind of address boundary at which a data segment is to start. It is not supported in BLISS–36 or BLISS–16; hence, it is nontransportable. Suitable default alignments are available dependent on the size of the scalar.

The weak attribute (WEAK) is a VMS-specific attribute and is not supported by BLISS–36 or BLISS–16; therefore, it cannot be used in a transportable program.

These guidelines are relatively simple and can relieve your concerns as to how the compiler will actually allocate the program's data. There is often very little reason to specify an allocation unit or any attributes. The default values are almost always sufficient.

There will undoubtedly be cases where it is impossible to avoid using one or more of the above attributes. In fact, it may be desirable to take advantage of a specific machine feature. In such cases, adhere to the following guideline:

- Conditionalize or heavily comment the use of declarations that are nontransportable.

This is an "escape-hatch" guideline that must be used sparingly and only where justified; frequent use will merely produce more code, which will have to be rewritten when the progam is transported to another machine.

6.4.2 Data Addresses

This section discusses address values and address value calculations. The section first presents problems resulting from using an address or an address calculation as a value, and then offers transportable solutions to some of these problems. This is followed by a discussion of the need for address forms of BLISS relational operators and control expressions and how and when to use them. Finally, some important differences in interpreting BLISS–10 and BLISS–36 address values are discussed.

6.4.2.1 Addresses and Address Calculations

The value of an undotted variable name in BLISS is an address. In most cases, this address value is used only for the simple fetching and storing of data. When address values are used for other purposes, we must be concerned with the portability of an address or an address calculation. The term "address calculation" means any arithmetic operations performed on address values. an address calculation. The term "address calculation" means any arithmetic operations performed on address values. The primary reason for this concern is the different sizes (in bits) of addressable units, addresses, and BLISS values (machine words) on the three machines. For convenience in writing transportable programs, these size values have been parameterized and are now predeclared literals. A table of their values can be found in Section 6.3.1.

To see how these size differences can have an effect on writing transportable programs, consider a common type of address expression that computes an address value from a base (a pointer or an address) and an offset. For example:

```
... base + index ...
```

Now consider the following BLISS assignment expression using this form of address calculation:

```
OWN
   ELEMENT_2;
.
.
.
ELEMENT_2 = .(INPUT_RECORD + 1);
```

The intent of the code is to access the contents of the second value in the data segment named INPUT_RECORD and to place that value in an area pointed to by ELEMENT_2. However, as will be shown, the effect of this code differs on each machine.

Adding 1 to an address (in this case, INPUT_RECORD) computes the address of the next addressable unit on the machine. For BLISS–32 and BLISS–16, the next addressable unit is the address of the next byte (8 bits), while for BLISS–36 the next addressable unit is the address of the next word (36 bits). Because the addressable units and the resultant values differ, the code segment does not reflect a transportable expression.

Therefore, adhere to the following guideline:

- When a complex address calculation is not an intrinsic part of the algorithm being coded, do not write it outside of a structure declaration.

There is a way, however, of making such an address calculation transportable. It involves the use of the values of the predeclared literals. In the last example, if the index had been 4 in BLISS–32 or 2 in BLISS–16, then in each case the next word would have been accessed.

A multiplier that will have a value of 4 in BLISS–32, 2 in BLISS–16, and 1 in BLISS–36 is needed. Such a multiplier exists as a predeclared literal. Its definition is %BPVAL/%BPUNIT, and it is called %UPVAL.

Using this literal in the example yields the following:

```
ELEMENT_2 =
   .(INPUT_RECORD + 1 * %UPVAL);
```

This last example raises an interesting point. If an address calculation of this form is used, then it is very likely that the data segment should have had a structure such as a VECTOR, BLOCK, or BLOCKVECTOR associated with it. The last example could have then been written as follows:

```
OWN

   INPUT_RECORD:
       FLEX_VECTOR[RECORD_SIZE,%BPVAL],
   ELEMENT_2;
.
.
.
ELEMENT_2 = .INPUT_RECORD[1];
```

The transportable structure FLEX_VECTOR and a more thorough discussion of structures can be found in Section 6.4.5.

6.4.2.2 Relational Operators and Control Expressions

The previous example illustrated using address values in the context of computations. Other common uses of addresses are in comparisons (for example, testing for equality) and as indices in loop and select expressions. Using address values in these contexts points to another set of differences found among the three machines.

In BLISS–32 and BLISS–16, addresses occupy a full word (%BPADDR equals %BPVAL) and unsigned integer comparisons must be performed. However, in BLISS–36, addresses are smaller than the machine word (18 versus 36 bits) and signed integer operations are performed for efficiency reasons.

For example, consider this relational test of address values:

```
... ADDRESS_1 LSS ADDRESS_2 ...
```

This test requires two different interpretations. This expression would evaluate correctly on DECsystem-10/20 systems. But on VAX and PDP–11 machines, the test would have to have been written as follows:

```
... ADDRESS_1 LSSU ADDRESS_2 ...
```

Another type of relational operator, designed specifically for address values, is needed. Such operators exist and are referred to as address relational operators. BLISS–36, BLISS–16, and BLISS–32 have a full set (for example, LSSA, EQLA, and so on), which support address comparisons.

In BLISS–16 and BLISS–32, address relational operators are equivalent to unsigned relational operators. In BLISS–36, address relational operators are equivalent to signed relational operators. However, for all practical purposes, you need not be concerned with this, because “equivalencing” permits address comparisons to be performed correctly across architectures. In addition, there are address forms of the SELECT (SELECTA), SELECTONE (SELECTONEA), INCR (INCRA), and DECR (DECRA) control expressions. The following guidelines establish a usage for these operators and control expressions:

- If address values are to be compared, use the address form of the relational operators.
- If an address is used as an index in a SELECT, SELECTONE, INCR, or DECR expression, use the address form of these control expressions.

Violating either of these guidelines can have unpredictable results.

6.4.2.3 BLISS–10 Addresses Versus BLISS–36 Addresses

There is a fundamental conceptual change from BLISS–10 to BLISS–36 in the defined value of a name. BLISS–10 defines the value of a data segment name to be a byte pointer consisting of the address value in the low half of a word, and position and size values of 0 and 36 in the high half of the word. BLISS–36, however, defines the value as simply the address in the low half and zeros in the high half. This change was made solely for reasons of transportability, because it allows BLISS to assign uniform semantics to an address.

The fetch and assignment operators are redefined to use only the address part of a value. For example:

```
Y = .X;
Y = F(.Y) + 2;
```

These expressions are the same in both BLISS–10 and BLISS–36, but the following expression assigns a different value to Y in BLISS–36 and in BLISS–10.

```
Y = X;
```

Field selectors are still available but must be thought of as extended operands to the fetch and assignment operators instead of as value producing operators applied to a name. For example:

```
Y<0,18> = .X<3,7>;
```

The meaning of this expression is unchanged, but the following expression is invalid:

```
Y = X<3,7>;
```

Moreover, it is strongly recommended that field selectors never appear outside of a structure declaration, because position and size are apt to be highly machine dependent. A more thorough discussion can be found in Section 6.4.5.

6.4.3 Data Character Sequences

This section discusses the use of character sequences (strings) in BLISS programs. Historically, there has been no consistent method for transportably dealing with strings and the functions operating upon them. Ad hoc string functions have been the rule, having been implemented by individuals or projects to suit their particular needs. This section begins by looking at quoted strings in two different contexts. Transportability problems associated with quoted strings and guidelines for their use are also described.

Quoted strings are used in two different contexts:

- As values (integers)
- As character strings

6.4.3.1 Quoted Strings Used as Numeric Values

Using quoted strings as values (in assignments and comparisons) illustrates the problem of differing representations on differing architectures. For example, consider the following code sequence:

```
OWN
   CHAR_1;     ! To hold a literal
CHAR_1 = 'ONE';
```

A natural interpretation for BLISS–32 to use is that one longword would be allocated and the three characters would be assigned to increasing byte addresses within the longword. In memory, the value of CHAR_1 would have the following representation:

```
CHAR_1:   / 00 E N O / (32)
```

BLISS–16 would not allow this assignment because only two ASCII characters are allowed per string literal. This restriction arises from the fact that BLISS–16 works with a maximum of 16-bit values, and three 8-bit ASCII characters require 24 bits.

On the DECsystem-10/20 a word would be allocated and the characters would be positioned starting at the high-order end of the word. Thus the string literal would have the following representation in memory:

```
CHAR_1:   / O N E 00 00 0 / (36)
```

Even if the DECsystem-10/20 string literal had been right justified in the word, it still would not equal the VAX representation numerically. So, in fact, the following would not be transportable:

```
WRITE_INTEGER('ABC' );
```

This is because 'ABC' is invalid syntax in BLISS–16, has the value –33543847936 in BLISS–36, and the value 4276803 in BLISS–32.

Based on these problems with representation, the first guideline is as follows:

- Do not use string literals as numeric values.

In those cases where it is necessary to perform a numeric operation (for example, a comparison) with a character as an argument, you must use the %C form of numeric literal. This literal takes one character as its argument and returns as a value the integer index in the collating sequence of the ASCII character set, so that the following is true:

```
%C'B' = %X'42' = 66
```

The %C notation was introduced to standardize the interpretation of a quoted character across all possible ASCII-based environments. %C'quoted-character' can be thought of as right-adjusting the character in a bit string containing %BPVAL bits.

6.4.3.2 Quoted Strings Used as Character Strings

The necessity of using more than one character in a literal leads to the other situation in which quoted strings are used: as character strings.

To facilitate the allocation, comparison, and manipulation of character sequences, a built-in character-handling package has been constructed as part of the BLISS language. It has been implemented in BLISS–32, BLISS–36, and BLISS–16.

These built-in functions provide a complete and powerful set of operations on characters. The next guideline is as follows:

- Use the built-in character-handling package when allocating and operating on character sequences. This is the only way you can guarantee the portability of strings and string operations.

A more detailed description of these functions can be found in the *BLISS Language Reference Manual*.

6.4.4 PLITs and Initialization

This section is primarily concerned with the uses of the PLIT (Pointer to Literal) and the UPLIT (Uncounted Pointer to Literal). First, there is general discussion of PLITs and the contexts in which they often appear. A presentation of how scalar PLIT items should be used follows. Next, the problems involved in using string literals in PLITs and suggested guidelines for their use are presented. Finally, the use of PLITs to initialize data segments is illustrated by the development of a transportable table of values.

6.4.4.1 PLITs in General

Because BLISS values are a maximum of a machine word in length, any literal that requires more than a word for its value needs a separate mechanism, and that mechanism is the PLIT. Hence, PLITs are a means for defining references to constants that are longer than one word. PLITs are often used to initialize data segments (such as tables) and are used to define the arguments for routine calls.

PLITs themselves are transportable; however, their constituent elements and their machine representation are not always transportable.

A PLIT consists of one or more values (PLIT items). PLIT items can be strings, numeric constants, address constants, or any combination of these last three, providing that the value of each is known before execution time.

6.4.4.2 Scalar PLIT Items

The first transportability problem that might be encountered with the use of PLITs is in specifying scalar PLIT items. As with any other declaration of scalar items (pointers, integers, addresses, and so on) it is possible to define them with an allocation-unit attribute. For example, in BLISS–32, machine-specific sizes (such as BYTE and LONG) can be specified. Thus, the following example is nontransportable and, in fact, will not compile on BLISS–36 or BLISS–16:

```
BIND
    Q1 = PLIT BYTE(1, 2, 3, LONG (-4));
```

This last example provides the first PLIT guideline:

- Do not use allocation units in the specification of a PLIT or PLIT item.

This last guideline is necessary because of the differences in the sizes of words on the three machines, a feature of the architectures. A discussion of the role of machine architectures in the transportability of data can be found in Section 6.4.1. Further guidelines are presented in Section 6.4.4.5.

6.4.4.3 String Literal PLIT Items

The next guideline is based on the representation of PLITs in memory. Specifically the problem is encountered when scalar and string PLIT items appear in the same PLIT.

The difficulty arises primarily from the representation of characters on the different machines. A more thorough discussion of character representation can be found in Section 6.4.3.

Use caution when you use strings as items in PLITs. For example, it may be necessary to specify a PLIT that consists of two elements: a 5-character string and an address of a routine. Suppose it is specified as follows:

```
BIND
    CONABC = PLIT('ABCDE', ABC_ROUT);
```

Then the VAX representation is as follows:

```
CONABC:                 / D C B A /  (32)
                        /       E /  (32)
                        / address /  (32)
```

On the PDP–11, it is as follows:

```
CONABC:                    / B A /  (16)
                           / D C /  (16)
                           /   E /  (16)
                      / address   / (16)
```

The DECsystem-10/20 representation is as follows:

```
CONABC:               / A B C D E /  (36)
                      / address   /  (36)
```

The three PLITs are not equivalent. Three longwords are required for the BLISS–32 representation, four words are needed for BLISS–16, and two words are needed for the BLISS–36 representation. There is a problem if the second element of this PLIT is to be accessed by the use of an address offset. For example, the second element (the address) is accessed by the following expression in the BLISS–36 version, but not in the BLISS–32 or BLISS–16 versions:

```
... CONABC + 1 ...
```

For the BLISS–32 version, the access expression is as follows:

```
... CONABC + 8 ...
```

For BLISS–16, it would have to be as follows:

```
... CONABC + 6 ...
```

Taking a data segment's base address and adding to it an offset (as in this case) is particularly sensitive to transportability. A discussion on the use of addresses can be found in Section 6.4.2.

This section on addresses suggests the use of the literal, %UPVAL, to ensure some degree of transportability. Its value is the number of addressable units per BLISS value or machine word. As already discussed, in BLISS–32, the literal equals 4; in BLISS–16, it is 2; and in BLISS–36, its value is 1.

Multiplying an offset by this value can, in some cases, ensure an address calculation that will be transportable. So to access the second element in the above PLIT, you would write the following:

```
... CONABC + 1*%UPVAL ...
```

But this will not work for the VAX representation. An offset value of 8 is needed because the string occupies two BLISS values. The situation is similar for the 11 version, where the string occupies 3 words and would need a offset value of 6, not 2.

The problem with this particular example (and, in general, with strings in PLITs) is not in using a string literal but in its position within the PLIT. Because the number of characters that will fit in a BLISS value differs on all three machines, the placement of a string in a PLIT will very often result in different displacements for the remaining PLIT items.

There is a relatively simple solution to this problem:

- In a PLIT there can only be a maximum of one string literal, and that literal must be the last item in a PLIT.

Following this guideline, the example should have been written as follows:

```
BIND
    CONABC = PLIT(ABC_ROUT, 'ABCDE');
```

Thus, the following expression would have resulted in the address of the second element in the PLIT (in this case the string).

```
... CONABC + 1*%UPVAL ...
```

6.4.4.4 Initialization Example

As mentioned in the beginning of this section, PLITs are often used to initialize data segments such as tables. You can initialize a data segment allocated by an OWN or GLOBAL declaration by using the INITIAL attribute. The INITIAL attribute specifies the initial values and consists of a list of PLIT items.

The following example uses a table of employee data to show how to initialize data in a transportable way. Information on each employee consists of three elements: an employee number, a cost center number and the employee's name. (The employee's name is a fixed length, 5-character field.)

For example, a line of the table contains the following information:

```
345     201     SMITH
```

Converting this line into a list of PLIT items that conform to this section's guidelines results in the following:

```
(345, 201, 'SMITH')
```

Notice that no allocation units were specified and that the character string was specified last. This line is now used to initialize a small table of only one line. The table has the built-in BLOCKVECTOR structure attribute. The table declaration is as follows:

```
OWN
    TABLE:
BLOCKVECTOR[1,3]
INITIAL(
345,
201,
'SMITH'
);
```

This definition works well in BLISS–36. That is, three words are allocated for TABLE. The first word is initialized with the employee number; the second word with the cost center; and the third with the name. However, the declaration is incorrect for BLISS–32 or BLISS–16 because not enough storage is allocated for all of the initial values. BLISS–32 requires four longwords and BLISS–16, five words.

To solve this problem, you determine the number of BLISS values that are needed for the character string on each machine. You then use the CH$ALLOCATION function from the character handling package to return this value. The CH$ALLOCATION function takes as an argument the number of characters to be allocated and returns the number of words needed to represent a string of this length. You can use this value as an allocation actual in the table definition, as follows:

```
OWN
    TABLE:  BLOCKVECTOR[1,2 + CH$ALLOCATION(5)]
INITIAL(
345,
201,
'SMITH'
);
```

The declaration is now transportable. By using the CH$ALLOCATION function you can ensure that enough words will be allocated on each machine. No rewriting is necessary.

You are free to add other lines to the table and not be concerned with the representation or allocation of the data. Here is a larger example of the same kind of table.

```
                    ...
                    ...
                    ...

!+
!       Table Parameters
!--

LITERAL
    NO_EMPLOYEES = 2,
    EMP_NAME_SIZE = 25,
    EMP_REC_SIZE = 2 +
            CH$ALLOCATION(EMP_NAME_SIZE);

!+
!       Employee Name Padding Macro
!--

MACRO
    NAME_PAD(NAME) =
        %EXACTSTRING (EMP_NAME_SIZE, 0, NAME)%;

!+
!       Employee Information Table
!
!       Size: NO_EMPLOYEES * EMP_REC_SIZE
!--

OWN
    EMP_TABLE:
        BLOCKVECTOR[NO_EMPLOYEES, EMP_REC_SIZE]
        INITIAL(
                345,
                201,
                NAME_PAD('SMITH PETER'),

                207,
                345,
                NAME_PAD('JONES PENNY')

                );
        ...
        ...
        ...
```

The literals serve to parameterize certain values that are subject to change. The literal EMP_REC_SIZE has as its value the number of words needed for a table entry. The character sequence function, CH$ALLOCATION, returns the number of words needed for EMP_NAME_SIZE characters.

The macro will, based on the length of the employee name argument (NAME), generate zero-filled words to pad out the name field. Thus, you are assured of the same number of words being initialized for each employee name, no matter what its size might be. This is important because storage is allocated according to the fixed length of a character field (employee name). The actual string length may, of course, be less than that value.

This last example was developed with the specification that the employee name field was fixed in length (EMP_NAME_SIZE). However, if you want the table to hold variable length names (that is, you want to allocate only enough storage to hold the table data, not the maximum amount), the table structure developed above will not work because it is predicated upon the constant size of the name field. If you were to use variable length character strings, either too much or not enough storage would be allocated. There would be no consistent way of accessing the employee name because you would not know where each name would start. You could, if you knew the length of every employee name, determine in advance the number of words needed but this is not a practical solution.

One transportable solution is to remove the character string from the table and replace it with a pointer to the string. The character package has a function (CH$PTR) that constructs a pointer to a character sequence. As an added benefit, this pointer can be used as an argument to the functions in the character package. The cost of this technique is the addition of an extra word (the character sequence pointer) for each table entry. The length of the name may also be stored in the table. Here is a typical example, again based on the employee table:

```
!+
!       Table Parameters
!--

LITERAL
    NO_EMPLOYEES = 2,
    EMP_REC_SIZE = 4;

!+
!       Macro to construct a CS-pointer to employee name
!--

MACRO
    NAME_PTR(NAME) =
       CH$PTR(UPLIT(NAME )), %CHARCOUNT (NAME) %;

!+
!       Employee Information Table
!
!       Size: NO_EMPLOYEES * EMP_REC_SIZE
!--

OWN
    EMP_TABLE:
        BLOCKVECTOR[NO_EMPLOYEES, EMP_REC_SIZE]
        INITIAL(
                345,
                201,
                NAME_PTR('SMITH PETER'),

                207,
                345,
                NAME_PTR('JONES PENNY')

                );
        ...
        ...
        ...
```

6.4.4.5 Initializing Packed Data

In this section we will discuss some transportability considerations involved in initializing of packed data. Packed data means that for data values v1, v2, ..., vn with bit positions p1, p2, ..., pn and bit sizes of s1, s2, ..., sn, respectively, the value of the PLIT item will be represented by the following expression:

```
v1^p1 OR v2^p2 OR ... OR vn^pn
```

where

$$max(p1, p2, ..., pn) < s1 + s2 + ... + sn \leq$$

for all i

$$-2 * *si \leq vi \leq 2 * *(si - 1)$$

The OR operator could be replaced by the addition operator (+), but the result would be different if, by accident, there were overlapping values. Note that the packing of data in a transportable manner is dependent on the value of %BPVAL.

The following is an illustration of the initialization of packed data obtained by modifying the employee table example developed above. When a field within a block is accessed, it is a common practice to associate each field reference (that is, offset, position, and size) with a field name. So, for example, the field names for the original employee table would look like the following:

```
FIELD EMP =
    SET
    EMP_ID = [0,0,%BPVAL,0],
    EMP_COST_CEN = [1,0,%BPVAL,0],
    EMP_NAME_PTR = [2,0,%BPVAL,0];
    TES;
```

You can use these field names in developing an initialization macro, by using parametric values. This is another example of how you can use parameterization as a key technique in writing transportable code.

If the number of bits needed to represent the values of EMP_ID and EMP_COST_CEN were each known not to exceed 16, you could pack these two fields into one BLISS value in BLISS–32 and BLISS–36. In BLISS–16 the definition of the employee table, as it now stands, would allocate only 16 bits for each field, since %BPVAL equals 16. In BLISS–36, an 18-bit size for these two fields would be chosen, because you know that DECsystem-10/20 systems have instructions that operate efficiently on halfwords.

If the interest is only in transporting BLISS–36 and BLISS–32, the field declaration would look like the following:

```
FIELD EMP =
    SET
    EMP_ID = [0,0,%BPVAL/2,0],
    EMP_COST_CEN = [0,%BPVAL/2,%BPVAL/2,0],
    EMP_NAME_PTR = [1,0,%BPVAL,0];
    TES;
```

Based on these declarations, you can design a macro that will take as arguments the initial values and then do the proper packing:

```
MACRO

    EMP_INITIAL(ID,CC,NAME)[] =
        ID^%FIELDEXPAND(EMP_ID,2) OR
        CC^%FIELDEXPAND(EMP_COST_CEN,2) ,
        NAME_PTR (NAME^%FIELDEXPAND(EMP_NAME_PTR, 2)) %;
```

The lexical function %FIELDEXPAND simply extracts the position parameter of the field name. The initialization macro, EMP_INITIAL, makes use of this shift value in packing the words. The goal here is to require the user to specify as arguments only the information needed to initialize the table, and not to specify information that is part of its representation. An example of using these macros to initialize packed data follows:

```
!+
!  Employee Field Reference macros
!--

FIELD EMP =
    SET
    EMP_ID  = [0,0,%BPVAL/2,0],
    EMP_COST_CEN  = [0,%BPVAL/2,%BPVAL/2,0],
    EMP_NAME_PTR  = [1,0,%BPVAL,0];
    TES;

MACRO
!+
!  Macro to create the shift value from the
!  position parameter of a field reference macro
!--

        SHIFT(X) = %FIELDEXPAND(X,2) %,

!+
!  Employee table initializing macro
!  Three values are required
!--

        EMP_INITIAL(ID,CC,NAME)[] =
              ID^SHIFT(EMP_ID) OR
              CC^SHIFT(EMP_COST_CEN),   ! First value

              NAME^SHIFT(EMP_NAME_PTR) %; ! Second value
!+
!   Employee table definition and initialization
!-

OWN
    EMP_TABLE:
        BLOCKVECTOR[NO_EMPLOYEES, EMP_LINE_SIZE]
        INITIAL(EMP_INITIAL(
                345,
                201,
               'SMITH PETER',

                207,
                345,
                'JONES PENNY'

                ));
```

Illustrated in the previous example is the parameterization of certain values such as field sizes. In transporting this program, you can derive benefits from localizing certain machine values as in the field definitions. This code is transportable between BLISS–32 and BLISS–36. To compile this program with the BLISS–16 compiler, a change to the field definitions is needed. The packing macros would no longer be needed, though they could be used for consistency. In that case, they would also need to be changed.

As a final example of initializing packed data, consider the DCB (data control block) BLOCK structure. (Details as to what DCB is and how it accesses data are discussed in the *BLISS Language Reference Manual*. Here, we are concerned only with initializing this type of structure.)

The DCB BLOCK consists of five fields. Four fields are packed into one word, their total combined size being 32 bits, and the fifth field, which is 32 bits in length, occupies another word.

In this case, it is possible to transport the DCB initialization very easily between BLISS–32 and BLISS–36. The reason is that the total number of bits required for each word does not exceed the value of %BPVAL for each machine. Hence, in this case at least, you do not have to modify the design of the BLOCK in any way. Typically, however, you would design the structure for each target machine. One method of doing this is by placing its definition in a REQUIRE file. It is preferable, however, to again use the parameterization technique, making use of the field reference macros as in the previous example.

The next example describes a method by which the DCB can be initialized. Using BLOCKVECTOR the structure is extended. Note that you can also transport the structure to BLISS–16 by making suitable changes to the field definitions and the packing macro. The only consideration might be whether the last field, DCB_E, did require a full 32 bits.

```
!+
!  DCB size parameters
!--

LITERAL
         DCB_NO_BLOCKS = total number of blocks,
         DCB_SIZE = size of a block;

!+
!  DCB Field Reference macros
!--

FIELD DCB =
     SET
     DCB_A = [0,0,8,0],
     DCB_B = [0,8,3,0],
     DCB_C = [0,11,5,0],
     DCB_D = [0,%BPVAL/2,%BPVAL/2,0],
     DCB_E = [1,0,%BPVAL,0];
     TES;
MACRO

!+
! Macro to create the shift value from the
! position parameter of a field reference macro
!--

        SHIFT(X) = %FIELDEXPAND(X, 2) %,

!+
! DCB initializing macro.
!  Five values are required.
!--

        DCB_INITIALIZE(A,B,C,D,E)[] =
                A^SHIFT(DCB_A) OR
                B^SHIFT(DCB_B) OR
                C^SHIFT(DCB_C) OR
                D^SHIFT(DCB_D) ,      ! first word

                E^SHIFT(DCB_E) %;     ! second word

!+
!  DCB Blockvector definition and initialization
!--
```

```
OWN
    DCB_AREA:
        BLOCKVECTOR[DCB_NO_BLOCKS, DCB_SIZE]
        INITIAL(
            DCB_INITIALIZE (
            1,2,3,4,                ! first word
            5,                      ! second word

            6,7,8,9,                ! first word
            10,                     ! second word
            ...
```

6.4.5 Structures and Field Selectors

Two BLISS constructs are discussed in this section: structures and field selectors. While using one does not necessarily imply using the other, for transportability reasons, field selector usage is confined to structure declarations. Hence, these two constructs are discussed together.

In the following general discussion of structures, it will be shown that a certain machine-specific feature of structures can be used in a transportable manner. The development of a transportable structure called FLEX_VECTOR is presented; field selectors are discussed next, followed by a description of a more general structure called GEN_VECTOR.

6.4.5.1 Structures

Structure declarations are sensitive to transportability in that you can specify parameters corresponding to characteristics of particular architectures. Also, in BLISS–32 the reserved words BYTE, WORD, LONG, SIGNED, and UNSIGNED have values of 1, 2, 4, 1, and 0, respectively, when used as structure actual parameters.

The ability to specify architecture-dependent information can be an advantage in developing transportable structure declarations. Later in this section, a structure will be developed that will use the UNIT parameter to gain a degree of transportability. The UNIT parameter specifies the number of addressable allocation units in one element of a homogeneous structure. This number will be used in determining the amount of storage that is to be allocated for each element of the structure.

As mentioned repeatedly in these guidelines, the prime transportability problem is differing machine architectures. The key to dealing with these differences is the parameterization, by the size of the machine word (%BPVAL), of the number of bits needed to hold an address (%BPADDR) and the number of bits occupied by the smallest addressable unit (%BPUNIT).

6.4.5.2 FLEX_VECTOR

An application of parametization is illustrated by FLEX_VECTOR, a structure that will, by default, allocate and access a vector consisting of only the smallest addressable units. If the default value given in the structure declaration is not used, the vector element size will be specified in terms of the number of bits. The existing VECTOR mechanism will not do this.

An example of its use would be to create a vector of 9-bit elements. The first decision that has to be made in its design is whether or not each element is to be exactly nine bits, or at least nine bits. For this example, you must choose the smallest natural unit whose size is greater than or equal to nine bits. Because there are no 9-bit conveniently addressable units on any of the machines, you have a choice of 8-, 16-, 32-, or 36-bit units.

Nine bits will fit in the only addressable unit on the DECsystem-10/20 systems: the word. On the PDP–11 you will need two bytes or a 16-bit word and on the VAX you will again need two bytes.

For a structure to do the allocation and also be transportable to the three systems requires some knowledge of machine architecture. This is where parameterization finds a key usage.

The predeclared literals have all the information you need. In fact, only one set of values is needed: bits per addressable unit (%BPUNIT).

The minimum necessary size of a vector element will be one of the allocation formals (UNIT). Other formals that will be needed are the number of elements (N), the index parameter (I) for accessing the vector, and an indication of whether or not the leftmost bit of an element is to be interpreted as a sign bit (EXT).

The access and allocation formal list for FLEX_VECTOR is as follows:

```
STRUCTURE
    FLEX_VECTOR[ I; N, UNIT = %BPUNIT, EXT = 1 ] =
```

Note that if you set UNIT equal to %BPUNIT, the default (if UNIT is not specified) is %BPUNIT.

The next step is to develop the formula for the structure-size expression. The expression makes use of the allocation formals UNIT and N and the value of %BPUNIT.

If UNIT were allowed to assume only values of integer multiples of %BPUNIT (that is, 1_*%BPUNIT, 2_*%BPUNIT, and so on), only a structure-size expression of the following form would be needed:

```
[ N * (UNIT) / %BPUNIT ]
```

Dividing the element size (UNIT) by %BPUNIT would give the size of each element in the vector in terms of an integer multiple. This value would then be multiplied by the number of elements to give the total size of the data to be allocated.

Suppose the structure needs to be more flexible in that it should be possible to specify any size element (within certain limits). The structure size must be slightly more complex:

```
[ N * ((UNIT + %BPUNIT - 1)) / %BPUNIT ]
```

The structure-size expression now computes enough %BPUNITs to hold the entire vector. You should try some values of UNIT, in order to see how the %BPUNIT expression evaluates.

The following NO_OF_UNITS subexpression is important to this particular structure in relation to transportability and flexibility:

```
(UNIT + %BPUNIT - 1) / %BPUNIT
```

The key to transporting the structure is in its knowledge of the value of a certain machine architectural parameter: bits per addressable unit. Because this particular expression makes use of this knowledge, it can adapt to any machine. The subexpression is used twice more in the structure-body expression.

The structure body is an address expression, which consists of the name of the structure (the base address) plus an offset based on the index I. In addition, a field selector is needed to access the proper number of bits at the calculated address.

The offset is simply the expression: NO_OF_UNITS multiplied by index I (remember that indices start at zero). The size parameter of the field selector is the expression NO_OF_UNITS multiplied by the size of an addressable unit (%BPUNIT).The structure body looks like the following:

```
(FLEX_VECTOR +
    I * ((UNIT + %BPUNIT - 1) / %BPUNIT))

<0,((UNIT + %BPUNIT - 1)/%BPUNIT)*%BPUNIT,EXT>;
```

The value of the position parameter in the field selector is a constant 0 since it always starts at an addressable boundary.

The following table shows the structure on the three machines for different values of UNIT:

Machine	Structure
VAX	
UNIT = 0	No storage FLEX_VECTOR<0,0,1>
UNIT = 1 to 8	[N * 1] Bytes (FLEX_VECTOR + I)<0,8,1>
UNIT = 9 to 16	[N * 2] Bytes (FLEX_VECTOR + I * 2)<0,16,1>
UNIT = 17 to 32	[N * 4] Bytes (FLEX_VECTOR + I * 4)<0,32,1>
PDP–11	
UNIT = 0 to 16	Same as VAX
DECsystem-10/20	
UNIT = 0	No storage (FLEX_VECTOR)<0,0,1>
UNIT = 1 to 36	[N] Words (FLEX_VECTOR + I)<0,36,1>

The table illustrates that if the default value for UNIT was set to %BPVAL this structure would be equivalent to a VECTOR of longwords on the VAX and a VECTOR of words on both DECsystem-10/20 and PDP–11 systems.

Elements in a data segment that has this particular structure attribute are accessed very efficiently because they are always on addressable boundaries. Also, they are always some multiple of an addressable unit in length.

If this structure were to access elements of exactly the size specified, the only change needed would be the size parameter of the field selector. This expression then becomes the following:

```
... FLEX_VECTOR<0, UNIT>;
```

This is a less efficient means of accessing data (when UNIT is not a multiple of %BPUNIT) because the compiler needs to generate field selecting instructions in the case of the VAX and DECsystem-10/20 machines and a series of masks and shifts for the PDP–11.

6.4.5.3 Field Selectors

In the last structure declaration, it was necessary to make use of a field selector. Using field selectors in a more general context are discussed in this section.

Using field selectors can be nontransportable because they use the value of the machine word size. The unrestricted use of field selectors can cause problems in a program when it is moved to another machine. These problems are best illustrated by the following table of restrictions on position (p) and size (s) for the three machines:

DECsytem-10/20	PDP–11	VAX
$0 \le p$	$0 \le p$	
$p + s \le 36$	$p + s \le 16$	
$0 \le s \le 36$	$0 \le s \le 16$	$0 \le s \le 32$

From the table, you can see that the PDP–11 is the most restrictive; the DECsystem-10/20 is moderately restrictive; and the VAX is the least restrictive.

To ensure the transportable use of field selectors, you would have to abide by the set of restrictions imposed in BLISS–16. These are restrictions imposed by the values of p and s. There is also a contextual restriction on the use of field selectors. The following guideline should be followed:

- Field selectors can appear only in the definition of user-defined structures.

Restricting the domain of field selectors to structures isolates their use. Field selectors should be isolated so that:

- Changes in data structure design are easier.
- Machine dependencies can easily be placed in REQUIRE files.
- Complex code making heavy use of the predeclared literals is limited to declarations.

6.4.5.4 GEN_VECTOR

GEN_VECTOR is a transportable structure that is also affected by the table of field selector value restrictions. Notice that FLEX_VECTOR does not attempt to pack data. Using the example of 9-bit elements, it is evident that there will be some wasting of bits—from seven bits on the PDP–11 and VAX to 27 on the DECsystem-10/20 systems.

A variation of FLEX_VECTOR can be developed to provide a certain degree of packing. For example, in the case of 9-bit elements it would be possible to pack at least four of them into a DECsystem-10/20 word and three into a VAX longword.

The GEN_VECTOR structure packs as many elements as possible into a BLISS value and so makes use of the machine-specific literal %BPVAL. However, since allocation is in terms of %BPUNIT, a literal is needed that has as a value the number of allocation units in a BLISS value. This literal has been predeclared for transportability reasons and has the name %UPVAL, and is defined as %BPVAL /%BPUNIT.

Elements do not cross word boundaries. This constraint exists because of the restrictions placed on the value of the position parameter of a DECsystem-10/20 and PDP–11 field selector. For the same reason elements cannot be longer than %BPVAL, as given in the table of field selector restrictions.

As in FLEX_VECTOR, the allocation expression of GEN_VECTOR needs to calculate the number of allocation units needed by the entire vector. This is again based on the number of elements (N) and the size of each and the size of each element (S); however, because the elements are packed, the expression is slightly more complicated.

The first value is the number of elements that will fit in a BLISS value. The following expression computes this value:

```
(%BPVAL/S)
```

Given this, to obtain the number of BLISS values or words needed for the entire vector, divide this value into N:

```
(N/(%BPVAL/S))
```

This is the total number of values needed. However, data are not allocated by words on both of the machines. Multiplying this value by %UPVAL results in the number of allocation units needed by the vector:

```
((N/(%BPVAL/S))*%UPVAL)
```

For clarity's sake and because this expression is again used, it is expressed as a macro with N and S as parameters:

```
MACRO
    WHOLE_VAL(N,S) =

         ((N/(%BPVAL/S))_*%UPVAL)%;
```

The name of the macro suggests that it calculates the number of whole words needed. If N were an integral multiple of the number of elements in a word, then this macro would be sufficient for allocation purposes.

Because this is not known in advance, another expression to calculate the number of allocation units needed for any remaining elements is needed. The number of elements left over is the remainder of the last division in this expression:

```
(N/(%BPVAL/S))
```

The MOD function calculates this value, as follows:

```
(N MOD (%BPVAL/S))
```

Multiplying this value by the size of each element gives the total number of bits that remain to be allocated:

```
(N MOD (%BPVAL/S)) * S
```

This value will always be strictly less than %BPVAL. For the same reasons outlined above, this expression will be expressed as a macro with N and S as parameters:

```
MACRO
    PART_VAL(N,S) =

        ((N MOD (%BPVAL/S)) * S)%;
```

PART_VAL computes the number of bits allocated in the last (partial) word.

Taking this value, adding an arbitrary factor, and then dividing by %BPUNIT gives us the number of allocation units needed for the remaining bits:

```
(PART_VAL(N,S) + %BPUNIT -1)/%BPUNIT
```

The total number of allocation units has been calculated and the structure allocation expression is as follows:

```
[WHOLE_VAL(N,S) +
(PART_VAL(N,S) + %BPUNIT - 1)/%BPUNIT]
```

As it works out, the structure-body expression for GEN_VECTOR is simple to write because of the expressions that have already been written.

The accessing of an element in GEN_VECTOR requires that an address offset be computed, which is then added to the name of the structure. The offset is some number of addressable units and is a function of the value of the index I. The expression calculating this number of units is the macro WHOLE_VAL. Thus, the first part of the accessing expression is as follows:

```
GEN_VECTOR + WHOLE_VAL(I,S)
```

Note that the macro was called with the index parameter I.

This expression results in the structure being aligned on some addressable boundary, but because the element may not begin at this point (that is, the element can be located somewhere within a unit %BPVAL bits in length), one more value is needed. That value is the position parameter of a field selector. The macro PART_VAL calculates this value based on the index I:

```
<PART_VAL(I,S),S,EXT>
```

The size parameter is the value S. The position parameter will be calculated at runtime, based on the value of the index I.

This completes the definition of GEN_VECTOR. The entire declaration is as follows:

```
STRUCTURE
    GEN_VECTOR[I;N,S,EXT=1] =

     [WHOLE_VAL(N,S) +
     (PART_VAL(N,S) + %BPUNIT - 1)/%BPUNIT]

     (GEN_VECTOR + WHOLE_VAL(I,S))

     <PART_VAL(I,S),S,EXT>;
```

You can compile this structure to see how it works in BLISS–16, BLISS–32, and BLISS–36.

6.4.5.5 Summary of Structures and Field Selectors

No claim is made that either of these two structures will solve all the problems associated with transporting vectors. Many such problems will have interesting and unique solutions. BLOCKS and BLOCKVECTORS have not been discussed, but it is hoped that you will get from the examples a feeling for the techniques involved in transporting structures.

There is no easy solution to transporting data structures. When developing data structures, you should consider the machines for which the program or system is targeted and make full use of predeclared literals, such as %BPUNIT.

This exercise in developing transportable structures has demonstrated the following two points:

- Parameterization
- Field selector usage

By parameterizing certain machine-specific values, and by taking full advantage of the powerful STRUCTURE mechanism, two transportable structures have been developed.

Although you can access odd (nonaddressable) units of data using field selectors, they should only be used in structure declarations.

7
Compiler Overview and Optimization Switches

This chapter provides an overview of the BLISS compiler organization and processing. The material presented here assumes that you have a general understanding of compiler theory and practice. It need not be understood for normal use of the BLISS language and compiler.

Some of the switches described in the *BLISS Language Reference Manual* provide specialized control over the processing of the compiler, especially in the area of optimization. This section provides the basis for a more detailed understanding of these switches. The switches that are described are as follows:

- CODE and NOCODE
- OPTIMIZE and NOOPTIMIZE
- OPTLEVEL
- SAFE and NOSAFE
- ZIP and NOZIP

7.1 Compiler Phases

The compiler is organized conceptually into seven major phases:

LEXSYN	Lexical and syntactic analysis
FLOW	Flow analysis
DELAY	Heuristics
TNBIND	Temporary name binding (register allocation)
CODE	Code generation
FINAL	Code stream optimization
OUTPUT	Object and listing file production

This division of the compiler into conceptual phases corresponds only approximately to the actual compiler. In some cases, a phase actually consists of two or more subphases. In other cases, phases are combined in the implementation; however, this level of detail is not important in the following discussion of the phases. The term “phase” should not be taken literally.

7.1.1 Lexical and Syntactic Analysis Phase

The lexical and syntactic phase (LEXSYN) performs the following functions:

- Reads the input files
- Divides the source character text into lexemes
- Identifies and performs lexical functions and macro expansions

- Parses the resulting input lexeme sequence and creates appropriate symbol table entries for declarations and tree representations for expressions

The BLISS compiler reads the source text once and uses it to create an internal representation of the module. In this sense, the BLISS compiler is a one-pass compiler. On the other hand, at the end of each (ordinary or global) routine definition, the remaining phases of the compiler are performed in turn to analyze and completely produce and output code for that entire routine. In this sense, the BLISS compiler is a multi-pass compiler.

If the NOCODE switch is specified, the compiler operates in "syntax-only" mode, in which the LEXSYN phase does not produce the tree representations for expressions and the later phases are not performed. If an error (as opposed to a warning) is detected and reported by the compiler, the compiler automatically enters syntax-only mode as if NOCODE had been specified.

Syntax-only mode is useful for initial checking of a newly created module. There is an important limitation in this mode, however, in that some errors cannot be detected because certain error detection and reporting can occur only in later phases of the compiler that are not performed.

The difference between an error and a warning diagnostic is based on the seriousness of the effect of the error on the internal representation of the program used by the compiler.

In most cases, the compiler can recover and proceed with normal compilation. This permits further errors, if any, to be detected and, in some cases, may permit the resulting object module to be used for execution time debugging before the source module is corrected and recompiled. Errors from which the compiler can continue normally are reported as warning diagnostics.

In some cases, the effect of a user error is to make the compiler's internal representation of the module inconsistent or otherwise unreliable for continued use. Such errors are reported as error diagnostics.

Depending on the circumstances, the same apparent user error (same diagnostic information) may be reported as a warning in one case but as an error in another.

7.1.2 Flow Analysis Phase

The flow analysis phase (FLOW) examines the internal tree representation of a complete routine and performs the following functions:

- Identifies expressions that appear more than once in the source but produce the same value (common subexpressions). Such expressions need be evaluated only once during execution and the result used several times, thereby saving execution time and code space.
- Identifies expressions contained in loops whose values will be the same on each iteration of the loop. Such expressions can be evaluated once before starting the loop and the result used during each iteration, thereby saving execution time.
- Identifies expressions that occur on all alternatives of the IF, CASE, and SELECTONE expressions. Such expressions can be evaluated once before or after the multiple alternatives, thereby saving code space.

More generally, the FLOW phase identifies possible alternative ways of evaluating a routine that might be more efficient in time or space, or both. Note that the next phase (DELAY) determines which alternative is actually used; the FLOW phase only identifies the possible choices.

If OPTLEVEL is specified with a value of 0 or 1, the flow analysis phase is skipped. A consequence of skipping flow analysis is that the OPTIMIZE and SAFE switches have no effect, because OPTIMIZE and SAFE control aspects of how flow analysis is done. However, if OPTLEVEL is specified with a value of 2 (the default) or 3, the flow analysis phase is performed and the OPTIMIZE and SAFE switches have effects that will be described.

To understand the effects of the OPTIMIZE and SAFE switches, it is first necessary to understand more about how flow analysis is performed.

7.1.2.1 Knowing When a Value Changes

One operator in BLISS, the assignment operator, can change the contents of a data segment. Routine calls can change the contents of data segments because they can contain assignments.

For each assignment, the compiler examines the left operand expression and attempts to determine the name of the data segment whose contents will be changed by the assignment. (The case where no name can be determined is considered below.) The compiler performs the same analysis for each actual parameter that appears in a routine call. In effect, the compiler treats each actual parameter as though it did appear as the left operand of an assignment. In addition to this, for each routine call the compiler determines the names of all OWN and GLOBAL data segments that the called routine might change and assumes that all of them are changed.

Machine-specific functions are treated as normal routine calls, except that the compiler has more detailed information about which parameters can cause changes and which cannot.

Several aspects of this analysis process are illustrated in the following examples. In the examples these declarations are assumed:

```
OWN
    X: VECTOR[10],
    Y,
    Z;
EXTERNAL ROUTINE
    F;
```

First, consider the following sequence of assignments:

```
I = 3;
Y = .X[.I];
X[7] = .X[.Y];
Z = .X[.I]+1;
```

In the third line, the assignment to X[7] is assumed to change all of the data segment identified by X. As a consequence, the compiler does not recognize the possible common subexpression .X[.I]. (However, note that the common subexpression X[.I], which computes the address of the Ith element of X, is recognized because the assignment to X cannot affect this value.)

In the above example, it may seem apparent exactly what part of X is changed, but in most cases it is difficult or impossible for the compiler to determine what part of a named data segment changes and what part does not change.

Another aspect is illustrated with this example:

```
X[11] = 11;
Y = .Z;
```

In the first line, the assignment to X[11] actually modifies the contents of Z. (Recall that X was declared as a vector of 10 elements numbered 0 through 9). The compiler analysis does not determine that storage other than the storage for X is being changed because the analysis is based completely on the names that occur in the expression. As a consequence, the compiler may inappropriately use the previous contents of Z in the assignment to Y. This would happen, for example, if the expression .Z were a common subexpression used frequently enough to result in the contents of Z being copied into a register for more efficient access.

Both of these examples emphasize the importance of the name used to reference storage in the analysis performed by FLOW.

Now consider the case where a name cannot be identified for the storage being changed. This is the case in the following example:

```
Z = F();
.Z = 3;
```

In the second line, no name of a data segment can be determined. In such a case, the compiler assumes (by default) that no named storage has changed. This assumption is justified because it is always the case that such indirect assignments are used to change the contents of the following:

- Dynamically created data structures that do not have names
- Data segments passed as parameters of routine calls that cannot be referenced in the called routine by the name used to allocate the storage

You can use the NOSAFE value of the /OPTIMIZE qualifier to override the default assumption described above (SAFE is the default). If NOSAFE is specified, the compiler assumes that indirect assignments change some named data segment. Because it is nearly impossible to identify the data segment that is changed, this assumption is guaranteed by the compiler by making the stronger assumption that all named data segments are changed.

7.1.2.2 Accounting for Changes

The BLISS language definition intentionally leaves unspecified the order of operand evaluation in operator expressions in order to permit maximum optimization by the BLISS compiler. For example:

```
F(X) + .X
```

This expression can be evaluated first, by calling F with the address X as a parameter; second, by fetching the contents of X; and finally, by performing the addition. It might also be evaluated first, by fetching the contents of X; second, by calling F; and so on. The compiler uses information about the entire routine in which the expression is contained to choose alternatives. Because the routine call F(X) may change the contents of X, the question becomes: When does the compiler account for the (potential) change? It does not make sense to account for this within the expression without precisely specifying the order of evaluation. It makes sense to account for changes only at points in the language where the order of evaluation is specified. Points at which changes are taken into account are called **mark points**. Mark points in BLISS are summarized in the following diagram, where an exclamation point (!) is used to point to the mark point within the language syntax on the subsequent line.

```
         !
BEGIN exp ;... END

      !        !        !
IF exp THEN exp ELSE exp

         !      !
WHILE exp DO exp

      !         !
DO exp WHILE exp

                  !      !      !      !
INCR name FROM exp TO exp BY exp DO exp

        !                                     !
CASE exp FROM ctce TO ctce OF SET [ ... ]: exp ;... TES

          !            !      !           !
SELECT exp OF SET [ exp TO exp ,... ]: exp ;... TES
```

The most common mark point in most programs is the semicolon, which separates expressions in a block or compound expression. For example:

```
BEGIN
Y = .X+2;
Z = .X+2+F(X);
W = .X+2
END
```

In the second line, the content of Y is changed. The compiler takes this change into account when the semicolon is encountered. In the third line, .X+2 computes the same value as .X+2 in the second line; thus, .X+2 is a common subexpression of the second and third lines. Also in the third line, the content of Z is changed and the call F(X) is considered to change the content of X. As discussed above, these changes are not taken into account until the semicolon is encountered. In the fourth line, .X+2 must be recomputed because of the change of the content of X in the third line; it is not a common subexpression with the previous occurrences.

The effect of the OPTIMIZE switch is now easily stated. If OPTIMIZE is specified (the default), full flow analysis is performed. If NOOPTIMIZE is specified, at every mark point all data segments are assumed to change. As a consequence, common subexpression values computed by one expression are not reused in later expressions, the value is computed again. Expressions that have a constant value within a loop are not computed once before the loop is started; the value is recomputed during each iteration of the loop. And similarly, other kinds of “code motion” optimizations are not performed.

However, specifying NOOPTIMIZE is not equivalent to specifying that no flow analysis is performed, because common subexpressions that occur between mark points are still detected. For example:

```
Y = (.X*2)+F(.X*2)
```

In this expression, subexpression .X*2 is computed once and the resulting value used twice, even when NOOPTIMIZE is specified.

7.1.3 Heuristic Phase

The heuristic phase (DELAY) further analyzes the routine to obtain general information about the routine. Some of this information is used by DELAY itself to make optimization decisions, and some is made available for use by later phases. DELAY performs the following functions:

- Evaluates the effectiveness of the alternatives identified by FLOW and chooses the best alternative. This analysis considers, for example, the number of occurrences of a common subexpression and the potential for using specialized operations available in the address parts of instructions (for example, indirection and indexing).
- Identifies sets of uncommon subexpressions that occur only once. These subexpressions should be computed in the same temporary register or memory location to maximize the use of 2-operand as opposed to 3-operand instructions.

None of the switches affect the operation of the DELAY phase.

7.1.4 Temporary Name Binding Phase

The temporary name binding phase (TNBIND) determines where each value computed during the execution of a routine should be allocated. This phase corresponds to what is sometimes called register allocation in other compilers. It is somewhat more general in that it considers and allocates user-declared local variables together with compiler-needed temporary locations in an integrated way.

TNBIND performs the following functions:

- Determines the lifetime (the first and last uses) of each temporary value in the routine.
- Estimates the difference in compiled code cost of allocating each temporary value in a register versus in memory (on the stack).
- Uses cost information to rank temporary values to determine the most important ones to be allocated in a register.
- Proceeding from most important to least important, allocates the temporary values. More than one temporary value may be allocated in the same location, provided their lifetimes do not overlap. (Thus, it is possible for a less important temporary to be allocated in a register even though a more important one is not—its shorter lifetime could permit it to “fit.”)

The measure of importance used (normally) by TNBIND is based completely on minimizing the overall size of the code generated for the entire routine.

The ZIP switch modifies the importance measure. If ZIP is specified, temporary values used within loops are given increased importance. The greater the degree of loop nesting, the greater the importance. Thus, temporary values used in loops become more likely to be allocated in registers. As a consequence, code within a loop tends to execute faster, even though the overall size of the routine can become larger.

7.1.5 Code Generation Phase

The code generator phase (CODE) processes the tree and generates instructions. Because the allocation for each operand of a node and the result location of each node have already been determined by TNBIND, CODE selects the locally best code sequence consistent with those requirements.

None of the compilation switches affect the operation of the CODE phase.

7.1.6 Code Stream Optimization Phase

The code stream optimization phase (FINAL) processes the code stream produced by CODE and makes further optimizations at the machine code level. The following optimizations are performed:

- Peephole optimization. One sequence of instructions is replaced with an equivalent shorter sequence.
- Test elimination. TST instructions used for conditional branching may be deleted if the condition codes needed by the branch instruction are appropriately set by the instructions that precede the TST.
- Jump/Branch resolution. Branch instructions are reduced to their shortest size consistent with the actual range of the needed branch displacement.
- Branch-chaining. If a short branch instruction is not able to reach the desired location but can reach another branch instruction that goes to the desired location, a chain of branches is used to minimize code size.
- Cross-jumping. Identical sequences of code that flow into a common instruction are merged into a single sequence.

You can use the OPTLEVEL switch to eliminate some of these optimizations. The result is code that more clearly follows the organization of the source program. This may be helpful during debugging or when the generated code must be understood in detail.

7.1.7 Output File Production Phase

The output file production phase (OUTPUT) transforms the code stream into object module format and outputs it to the object file. It also formats and outputs the listing file information.

If the DEBUG switch is specified, symbol table information for use by the DEBUG system utility is included in the object module. If NODEBUG is specified (the default), no symbol table information is produced.

7.2 Summary of Switch Effects

The previous sections have described the phases of the compiler and the switches that affect each of those phases. This section summarizes the effects of each switch throughout the compiler.

Switch Name	Phase	Effect
CODE	LEXSYN	NOCODE specifies syntax-only processing; the other phases are not invoked.
OPTIMIZE	FLOW	If flow analysis is performed, NOOPTIMIZE prevents optimization across mark points.

Switch Name	Phase	Effect
OPTLEVEL	FLOW	At levels 0 and 1, flow analysis is not performed.
	FINAL	At levels 0 and 1, cross-jumping and branch chaining are not performed. At level 0, peephole optimizations that are not adjacent are not performed.
SAFE	FLOW	If flow analysis is performed, NOSAFE specifies that indirect changes are assumed to change all storage.
ZIP	TNBIND	ZIP specifies that data segments used in loops are to be given increased importance in determining register allocation.

The OPTLEVEL switch is a composite switch that includes appropriate settings of the other switches in an ordered way. It can be specified in the command line or module head, or both. The rule applied to determine which switch setting has effect is that the most recent switch setting specified has effect allowing the other switches (SAFE, OPTIMIZE, and so on) to override OPTLEVEL at any time. In a compilation of more than one module, each module begins with the setting defined in the command line and OPTLEVEL=2 if OPTLEVEL has not been specified in the command line.

The optimizations performed at each setting of the OPTLEVEL switch are as follows:

1. *Common subexpression detection
2. *Code motion out of loops, and so on
3. Targetting/preferencing to temporaries
4. Cross jumping
5. Multiple RETs (instead of only one return point)
6. Peepholes
7. TSTx elimination
8. Scans for ()+, −() forms
9. Scans for PUSHR, and so on
10. Scans for cheaper form of addressing
11. Branch chaining
12. *SAFE optimizations
13. *OPTIMIZE over mark points (for example,;)
14. *ZIP speed/space trade off (faster but possibly larger)

The following table relates the optimizations to the OPTLEVEL settings:

	OPTLEVEL			
	0	1	2	3
1*			x	x
2*			x	x
3	x	x	x	+

	OPTLEVEL			
4			x	x
5		x	x	x
6	r	x	x	x
7	x	x	x	x
8	x	x	x	x
9	x	x	x	x
10	x	x	x	x
11			x	x
12*			x	x
13*			x	x
14*				x

Key to OPTLEVEL Settings

*—Another switch can control this optimization separately
x—Allowable optimization
+—Allowed (with increased freedom)
r—Allowed (with certain restrictions)

8

Tools, Libraries, and System Interfaces

A number of programming tools, libraries, and system interfaces are available for use by BLISS programmers. This chapter briefly describes what is available for use with BLISS–32 Version 4.3. Note that with the exception of the VAX Source Code Analyzer, the BLISS tools (utility programs) and system interfaces described here are not supported products at the time of this writing.

The residence locations specified for the various BLISS–related program and information files assume that these files have been installed on your system in their standard, as-released locations.

8.1 VAX Source Code Analyzer (SCA)

The VAX Source Code Analyzer (SCA) is a source code cross-reference and static analysis tool that is used to aid you in understanding large-scale software systems. SCA is integrated with the VAX Language-Sensitive Editor (LSE), but can also be used, in a reduced capacity, at the DCL level as a standalone tool. In either case, SCA libraries are created and analysis data files loaded, using SCA commands.

For details about LSE and SCA refer to Appendix G.

8.2 Tutorial Terminal Input/Output Package (TUTIO)

TUTIO.R32 is a BLISS REQUIRE file that contains some simple terminal I/O primitives. This package is normally used in conjunction with the BLISS self-paced study course, as outlined in the *BLISS Primer*, but can also be useful for writing short miscellaneous programs. To gain access to the elements of this package, insert the following line in your BLISS program:

```
REQUIRE 'SYS$LIBRARY:TUTIO';
```

A list of these primitives and their functions appears below. The following conventions are used in the descriptions:

char (character)
len (length in characters)
addr (memory address)
value (integer)
radix (integer)

- TTY_PUT_CHAR (char);—writes a character to the terminal.
- char=TTY_GET_CHAR ();—reads a character from the terminal.
- TTY_PUT_QUO ('quoted string');—writes a quoted string to the terminal.
- TTY_PUT_CRLF ();—writes a carriage return/line feed sequence to the terminal.
- TTY_PUT_ASCIZ (addr);—writes an ASCIZ string to the terminal.

- TTY_PUT_MSG (addr,len);—writes a string of ASCII characters to the terminal.
- TTY_PUT_INTEGER (value,radix,len);—writes an integer to the terminal.
- n = TTY_GET_LINE (addr,len);—reads a line from the terminal into a buffer and returns the number of characters read.

The TUTIO package functions only with a terminal; it does not work from a batch job or command procedure file.

8.3 VMS System Services Interface

System services are procedures used by the operating system to control resources available to processes, to provide for communication among processes, and to perform basic operating system functions, such as the coordination of input /output operations. Although most system services are used primarily by the operating system itself on behalf of logged-in users, many are available for general use and provide techniques that can be used in application programs.

The BLISS interface to the VMS system service routines is in SYS$LIBRARY:STARLET.REQ or SYS$LIBRARY:STARLET.L32. The primary function of the interface is to guarantee the proper number and sequence of arguments required for the system service calls.

With the BLISS interface macros, the syntax for invoking system services in BLISS is almost identical to the VAX MACRO keyword syntax. The major difference (aside from the fact that BLISS expressions appear in the parameter list) is that BLISS requires the macro parameter list to be enclosed in parentheses. (Following is an example of the BLISS–style usage.)

To use the interface, include the following declaration:

```
LIBRARY 'SYS$LIBRARY:STARLET'
```

System services act like routine calls and return a condition value as their result. The following code fragment checks for successful completion of the system service:

```
IF STATUS=$WAITFR(EFN=1)
THEN
    ...                 !successful wait
ELSE                    !invalid event flag
    SIGNAL(.STATUS);    !pass back failure
```

For detailed information on available services, see the *VAX/VMS System Services Reference Manual*.

8.3.1 Sample Program Using VMS System Services

The program shown below prints the time of day at the user's terminal. It shows the use of system services $GETTIM and $FAO and of the common run-time library routine LIB$PUT_OUTPUT.

```
MODULE SHOWTIME(IDENT='1-1' %TITLE'Print time', MAIN=TIMEOUT)=
BEGIN
LIBRARY 'SYS$LIBRARY:STARLET';      ! Defines System Services
```

```
OWN
    TIMEBUF: VECTOR[2],                 ! 64-bit system time
    MSGBUF: VECTOR[80,BYTE],            ! Output message buffer
    MSGDESC: BLOCK[8,BYTE] PRESET( [DSC$W_LENGTH]=80,
                          [DSC$A_POINTER]=MSGBUF)
BIND
    FMTDESC=%ASCID %STRING('At the tone, the time will be ',
            %CHAR(7), '!15%T' );
EXTERNAL ROUTINE
    LIB$PUT_OUTPUT: ADDRESSING_MODE(GENERAL);

ROUTINE TIMEOUT=
    BEGIN
    LOCAL

        RSLT: WORD;               ! Resultant string length

    $GETTIM( TIMADR=TIMEBUF );    ! Get time as 64-bit integer

    $FAOL( CTRSTR=FMTDESC,        ! Format Descriptor
        OUTLEN=RSLT,              ! Resultant length (only a word!)
        OUTBUF=MSGDESC,           ! Output buffer descriptor
        PRMLST= %REF(TIMEBUF));   ! Addr of 64-bit time block
    MSGDESC[0] = .RSLT;           ! Modify output descriptor
    LIB$PUT_OUTPUT( MSGDESC )     ! Return status
    END;
END ELUDOM
```

In the SHOWTIME example, LIB$PUT_OUTPUT is part of the VMSRTL shareable image; as such it is referenced by GENERAL addressing. The call on $FAOL uses %REF to make the PRMLST value the address of a pointer to the TIMEBUF buffer.

8.3.2 Common Errors in Using System Services

When you invoke system services, you may encounter a number of errors. Two of the more common errors occur when the $HIBER and $FAO services are invoked.

The $HIBER system service keyword macro has no arguments. Thus, it is invoked as follows:

```
$HIBER
```

Note the absence of any parenthesized argument list.

The OUTLEN keyword parameter of the $FAO and $FAOL system service is the address of a WORD (16-bit) value. A common error is to provide that parameter with the address of a fullword (32-bit) value rather than a WORD value. Thus, the 16 high-order bits always contain a variable, indeterminate value. Consequently, when the full 32-bit value is used rather than just the 16 low-order bits, unpredictable results occur.

You should guard against this problem in calls to other system services.

8.4 Record Management Services Interface

VAX Record Management Services (VAX RMS) is a collection of file and record-level I/O routines that enable user programs to process and manage data files. Also included as a part of VAX RMS is a set of BLISS macro definitions that facilitate control block initialization and the calling of VAX RMS control routines.

RMS definitions appear in SYS$LIBRARY:STARLET.REQ or SYS$LIBRARY:STARLET.L32. For a general description of VAX RMS programming, see the *VAX Record Management Services Reference Manual*.

8.4.1 Using VAX RMS Macros

STARLET.REQ provides three types of VAX RMS definitions:

1. Field definitions and constants used in VAX RMS control blocks
2. Static and dynamic control block initialization macros
3. VAX RMS routine-call keyword macros

Macros for defining FAB, RAB, and XAB blocks are provided. Each control block has macros for static or dynamic initialization and for uninitialized structures. For example:

$FAB	A keyword macro for static definition.
$FAB_INIT	A keyword macro for run-time dynamic initialization.
$FAB_DECL	A simple macro for FABs that require no initialization. Typically, this is used to map attributes onto a routine's formal parameter.

Keywords for these macros are identical to those used by the VAX RMS MACRO interface, except for those keywords that expect a 64-bit value. These keywords are handled as "key0" and "key1", each of which has a 32-bit value.

8.4.2 Sample Routine Using VAX RMS

The following sample program illustrates the use of the VAX RMS macros. The program opens a file named MYFILE.SRC and copies it to a file named MYFILE.LIS.

```
MODULE RMSTEST(IDENT='1-1' %TITLE'BLISS--RMS Example',MAIN=COPYIT)=
BEGIN
!++
! Sample program showing use of BLISS/RMS-32 Interface Macros
!--
LIBRARY 'SYS$LIBRARY:STARLET';          ! All definitions are here

OWN
    MYBUF:    VECTOR[CH$ALLOCATION(132)],   ! Input record buffer

    INFAB:  $FAB(FNM='MYFILE.SRC',          ! Source FAB
                 FAC=GET),

    OUTFAB: $FAB(FNM='MYFILE.LIS', RFM=VAR, ! Destination FAB
                 RAT=CR, FAC=PUT),

    INRAB:  $RAB(UBF=MYBUF, USZ=132,        ! RAB def'n: Locate Read,
                 ROP=<LOC,RAH>, FAB=INFAB), ! and Multibuffer I/O
    OUTRAB: $RAB(ROP=WBH, FAB=OUTFAB);      ! on input and output.

ROUTINE COPYIT=
    BEGIN
    LOCAL
       STS;      ! RMS service-completion code
```

```
    $OPEN( FAB=INFAB );     ! Open input file
    $CONNECT( RAB=INRAB );  ! and associate RAB.
    $CREATE( FAB=OUTFAB );  ! Create a new output file
    $CONNECT( RAB=OUTRAB ); ! and associate a RAB.
    !+
    ! Copy loop. Read records from input until
    ! EOF or error encountered. Write each record as it is read.
    !---
    WHILE ( STS=$GET( RAB=INRAB ) ) DO
         BEGIN
         OUTRAB[RAB$L_RBF] = .INRAB[RAB$L_RBF];  ! Copy record descr.
         OUTRAB[RAB$W_RSZ] = .INRAB[RAB$W_RSZ];  ! to output RAB and
         $PUT( RAB=OUTRAB )                      ! write the record.
         END;

    $CLOSE( FAB=INFAB );    ! Close the input
    $CLOSE( FAB=OUTFAB );   ! and output files.

    IF .STS EQL RMS$_EOF    ! Check for EOF on input
    THEN
        SS$_NORMAL          ! Tell System everything is OK, or
    ELSE
        .STS                ! Return the RMS error status.
    END;
END ELUDOM
```

8.5 Other VMS Interfaces

Two other VMS interfaces (LIB and TPAMAC) are available.

8.5.1 LIB Interface

LIB.L32 contains internal interfaces between BLISS–32 and the VMS operating system as well as everything in STARLET.L32. Much of LIB is normally useful only to an operating system programmer. Most users should use SYS$LIBRARY:STARLET.L32, a subset of LIB.

8.5.2 TPAMAC Interface

TPAMAC.REQ provides BLISS macros for LIB$TPARSE, a finite-state transition parser. The sample program presented in Example 8–1 closely parallels the example in the *VAX/VMS Run-Time Library Routines Reference Manual*.

Example 8–1 Sample TPARSE Program

```
MODULE CREATE_DIR(%TITLE 'Create Directory File' IDENT = 'X0000',
        MAIN=CREATE_DIR) =
BEGIN

!+
!
! This is a sample program that accepts and parses the command line
! of the CREATE/DIRECTORY command.
! This program contains the operating system
! call to acquire the command line from the command interpreter
! and parse it with TPARSE, leaving the necessary information
! in its global data base. The command line has the following format:
!
!       CREATE/DIR DEVICE:[MARANTZ.ACCOUNT.OLD]
!               /OWNER_UIC=[2437,25]
!               /ENTRIES=100
!               /PROTECTION=(SYSTEM:R,OWNER:RWED,GROUP:R,WORLD:R)
!
! The three qualifiers are optional. Alternatively, the command
! may take the form
!
!       CREATE/DIR DEVICE:[202,31]
!
! using any of the optional qualifiers.
!
!---

LIBRARY 'SYS$LIBRARY:STARLET';        ! Define CLI offsets
LIBRARY 'SYS$LIBRARY:CLIMAC';         !  and request blocks,
LIBRARY 'SYS$LIBRARY:TPAMAC';         !  and TPARSE stuff too.

FORWARD ROUTINE
        BLANKS_OFF,
        CHECK_UIC,
        STORE_NAME,
        MAKE_UIC,
        CREATE_DIR;
!
! Define parser flag bits for flags longword
!
LITERAL
    UIC_FLAG=          1,               ! /UIC seen
    ENTRIES_FLAG=      2,               ! /ENTRIES seen
    PROT_FLAG=         4;               ! /PROTECTION seen

!
! CLI request descriptor block to get the command line
!
OWN
    REQ_COMMAND: $CLIREQDESC( RQTYPE=GETCMD );
!
! TPARSE parameter block
!
OWN
    TPARSE_BLOCK: BLOCK[TPA$C_LENGTH0,BYTE]
                  PRESET( [TPA$L_COUNT]=TPA$K_COUNT0,   ! Longword count
                          [TPA$L_OPTIONS]= TPA$M_ABBREV ! Allow abbreviation
                                OR TPA$M_BLANKS);       ! Process spaces explicitly
```

(continued on next page)

Example 8–1 (Cont.) Sample TPARSE Program

```
!
! Parser Data Base
!
OWN
    PARSER_FLAGS,                       ! Keyword flags
    DEVICE_STRING:      VECTOR[2],      ! device string descriptor
    ENTRY_COUNT,                        ! space to preallocate
    FILE_PROTECT,                       ! directory file protection
    UIC_GROUP,                          ! temp for UIC group
    UIC_MEMBER,                         ! temp for UIC member
    UIC_OWNER,                          ! actual file owner UIC
    NAME_COUNT,                         ! number of directory names
    DIRNAME:      BLOCKVECTOR[8,2,LONG]; ! name descriptors 0-7

EXTERNAL ROUTINE
    SYS$CLI:    ADDRESSING_MODE(GENERAL),
    LIB$TPARSE: ADDRESSING_MODE(GENERAL);

%SBTTL 'Parser State Table'
$INIT_STATE( UFD_STATE, UFD_KEY );
!
! Read over the command line to the first blank in the command.
!
        $STATE( START,
          (TPA$_BLANK,          ,BLANKS_OFF),
          (TPA$_ANY,     START));

!
! Read device name string and trailing colon
!
        $STATE(,
          (TPA$_SYMBOL,        ,      ,      ,DEVICE_STRING));
        $STATE(,
          (':'));
!
! Read directory string, which is either a UIC string or a general
! directory string.
!
```

(continued on next page)

Example 8–1 (Cont.) Sample TPARSE Program

```
        $STATE(,
          ( (UIC),             ,MAKE_UIC),
          ( (NAME)));
!
! Scan for options until end of line is reached
!
        $STATE(OPTIONS,
          ( '/' ),
          ( TPA$_EOS,    TPA$_EXIT ) );
        $STATE(,
          ('OWNER_UIC', PARSE_UIC,      , UIC_FLAG,     PARSER_FLAGS),
          ('ENTRIES',   PARSE_ENTRIES,  , ENTRIES_FLAG, PARSER_FLAGS),
          ('PROTECTION',PARSE_PROT,, PROT_FLAG,
   PARSER_FLAGS) );
!
! Get file owner UIC
!
       $STATE( PARSE_UIC,
         (':'),
         ('='));
       $STATE(,
         ( (UIC),      OPTIONS ));
!
! Get number of directory entries
!
       $STATE( PARSE_ENTRIES,
         (':'),
         ('='));
       $STATE(,
         (TPA$_DECIMAL,OPTIONS,       ,        ,       ENTRY_COUNT));


!
! Get directory file protection. Note that the bit masks generate the
! protection in complement form. It will be uncomplemented by the main
! program
!
       $STATE( PARSE_PROT,
         (':'),
         ('='));
       $STATE(,
         ('('));

       $STATE( NEXT_PRO,
         ('SYSTEM',    SYPR ),
         ('OWNER',     OWPR ),
         ('GROUP',     GRPR ),
         ('WORLD',     WOPR ) );

       $STATE( SYPR,
         (':'),
         ('=') );

       $STATE(SYPRO,
         ('R',         SYPRO,  , %X'1',        FILE_PROTECT),
         ('W',         SYPRO,  , %X'2',        FILE_PROTECT),
         ('E',         SYPRO,  , %X'4',        FILE_PROTECT),
         ('D',         SYPRO,  , %X'8',        FILE_PROTECT),
         (TPA$_LAMBDA, ENDPRO) );

       $STATE(OWPR,
         (':'),
         ('='));
```

(continued on next page)

Example 8–1 (Cont.) Sample TPARSE Program

```
        $STATE(OWPRO,
          ('R',         OWPRO,  , %X'0010',     FILE_PROTECT),
          ('W',         OWPRO,  , %X'0020',     FILE_PROTECT),
          ('E',         OWPRO,  , %X'0040',     FILE_PROTECT),
          ('D',         OWPRO,  , %X'0080',     FILE_PROTECT),
          (TPA$_LAMBDA, ENDPRO) );

        $STATE(GRPR,
          (':'),
          ('='));

        $STATE(GRPRO,
          ('R',         GRPRO,  , %X'0100',     FILE_PROTECT),
          ('W',         GRPRO,  , %X'0200',     FILE_PROTECT),
          ('E',         GRPRO,  , %X'0400',     FILE_PROTECT),
          ('D',         GRPRO,  , %X'0800',     FILE_PROTECT),
          (TPA$_LAMBDA, ENDPRO) );

        $STATE(WOPR,
          (':'),
          ('='));

        $STATE(WOPRO,
          ('R',         WOPRO,  , %X'1000',     FILE_PROTECT),
          ('W',         WOPRO,  , %X'2000',     FILE_PROTECT),
          ('E',         WOPRO,  , %X'4000',     FILE_PROTECT),
          ('D',         WOPRO,  , %X'8000',     FILE_PROTECT),
          (TPA$_LAMBDA, ENDPRO) );

        $STATE(ENDPRO,
          (',',         NEXT_PRO),
          (')',         OPTIONS) );
!
! Subexpressions to parse a UIC string.
!
        $STATE( UIC,
          ('[') );
        $STATE(,
          (TPA$_OCTAL,          ,       ,       ,       UIC_GROUP) );
        $STATE(,
          (','));
        $STATE(,
          (TPA$_OCTAL,          ,       ,       ,       UIC_MEMBER) );
        $STATE(,
          (']',          TPA$_EXIT, CHECK_UIC) );
!
! Subexpressions to parse a general directory string
!
        $STATE( NAME,
          ('[') );
        $STATE( NAMEO,
          (TPA$_STRING,         , STORE_NAME) );
        $STATE(,
          ('.',         NAMEO ),
          (']',         TPA$_EXIT) );
!
! Note absence of $END_STATE macro.
!
%SBTTL 'Parser Action Routines'

ROUTINE BLANKS_OFF=
```

(continued on next page)

Example 8–1 (Cont.) Sample TPARSE Program

```
 !
 ! Shut off explicit blank processing after passing the command name.
 !
     BEGIN
     BUILTIN
AP;
     MAP
AP:    REF BLOCK[,BYTE];
     AP[TPA$V_BLANKS] = 0;
     1 ! Success always.

     END;

 ROUTINE CHECK_UIC =
 !
 ! Check the UIC for legal value range.
 !
     BEGIN
     BUILTIN
         AP;
     MAP
         AP:     REF BLOCK[,BYTE];

     IF .UIC_GROUP<16,16> NEQ 0 OR        ! UIC components are 16 bits only
        .UIC_MEMBER<16,16> NEQ 0
     THEN
         RETURN 0;

     UIC_OWNER<0,16> = .UIC_MEMBER<0,16>;
     UIC_OWNER<16,16> = .UIC_GROUP<0,16>;
     1
     END;

 ROUTINE STORE_NAME=
 !
 ! Store a directory name component
 !
     BEGIN
     BUILTIN
         AP;
     MAP
         AP:     REF BLOCK[,BYTE];

     IF .NAME_COUNT GEQ 3 THEN RETURN 0;         ! Maximum of 3 components

     DIRNAME[.NAME_COUNT,0,0,32,0] = .AP[TPA$L_TOKENCNT]; ! Store the
     DIRNAME[.NAME_COUNT,1,0,32,0] = .AP[TPA$L_TOKENPTR]; !  descriptor
     NAME_COUNT = .NAME_COUNT + 1;                        ! Set to next
                                                          !  free slot

     IF .AP[TPA$L_TOKENCNT] GTR 9 THEN RETURN 0;  ! Name too long

     1
     END;

 ROUTINE MAKE_UIC=
     BEGIN
     BUILTIN
         AP;
     MAP
         AP:     REF BLOCK[,BYTE];
     OWN
         UIC_STRING:     VECTOR[CH$ALLOCATION(6)];
```

(continued on next page)

Example 8–1 (Cont.) Sample TPARSE Program

```
    IF .UIC_GROUP<8,8> NEQ 0 OR                 ! Check UIC for byte values,
       .UIC_MEMBER<8,8> NEQ 0                   !  since UIC type directories
    THEN                                        !  are restricted to this form
        RETURN 0;

    DIRNAME[0,1,0,32,0] = UIC_STRING;           ! Point to string buffer
    $FAOL( CTRSTR=UPLIT(6,'!OB!OB'),            ! Convert UIC to octal string
           OUTBUF=DIRNAME,
           PRMLST=UIC_GROUP)
    END;

%SBTTL  'Main Program'

ROUTINE CREATE_DIR=
!+
!
! This is the main program of the CREATE/DIRECTORY utility. It gets
! the command line from the command interpreter and parses it with TPARSE.
!
!---
    BEGIN
    !
    ! Call the command interpreter to obtain the command line
    !
    SYS$CLI( REQ_COMMAND, 0, 0 );

    !
    ! Copy the input string descriptor into the TPARSE control block
    ! and call LIB$TPARSE. Note that impure storage is assumed to be zero.
    !
    TPARSE_BLOCK[TPA$L_STRINGCNT] = .REQ_COMMAND[CLI$W_RQSIZE];
    TPARSE_BLOCK[TPA$L_STRINGPTR] = .REQ_COMMAND[CLI$A_RQADDR];

    IF NOT LIB$TPARSE( TPARSE_BLOCK, UFD_STATE, UFD_KEY )
    THEN
        RETURN SS$_ABORT;

    !
    ! Parsing is complete
    !
                                        ! Process the command to create
                                        ! the appropriate directory.
     RETURN SS$_NORMAL
     END;                               ! Return Success to command interpreter
END ELUDOM
```

9 BLISS–32 Code Examples

This chapter provides useful examples of BLISS–32 coding techniques. These examples demonstrate the use of many BLISS language features.

9.1 Example 1: The ALPHA Program

This program prompts the user for a line of text, removes all but the uppercase characters from the line, then rearranges and prints the uppercase characters in alphabetical order. The algorithms chosen are not optimal, but they serve to demonstrate recommended style. The program consists of two routines (SORT_CHARS and ALPHA_SORT) and three included files.

The ALPHA program uses the following facilities:

- TUTIO—Tutorial Terminal Input/Output Package
- CELPKG—Cell Manipulation Primitives
- VMS RTL—Memory Management Routines (called within CELPKG)

Note that the ALPHA module is transportable but the TUTIO and CELPKG packages are not. TUTIO exists for target systems other than the VAX, while CELPKG can be written for other systems.

The following steps are used to run the ALPHA program:

1. Produce a library file for the CELPKG program:

   ```
   $ BLISS /LIB CELPKG
   ```

2. Compile the ALPHA program:

   ```
   $ BLISS ALPHA
   ```

3. Link the ALPHA program:

   ```
   $ LINK ALPHA
   ```

4. Run the ALPHA program:

   ```
   $ RUN ALPHA
   ```

5. As an example, respond to the input prompt as follows:

   ```
   INPUT: 12P4E6IQOY-='asFgjkW;'{ZXCBNM<>?ASDVRGHJKLTU RET
   OUTPUT:ABCDEFGHIJKLMNOPQRSTUVWXYZ
   INPUT:  CTRL/C
   $
   ```

The following sections contain the annotated ALPHA module and the cell-handling package (CELPKG).

9.1.1 Module ALPHA

```
MODULE ALPHA ( IDENT = 'x1-1', %TITLE'Sample Program'
                MAIN=ALPHA_SORT ) =
BEGIN

!++
! FACILITY:     Sample Program
!
! ABSTRACT:
!
!    This sample program prompts you for a line of text,
!    removes all but uppercase characters from the line,
!    and prints the uppercase characters in alphabetical
!    order. The program is terminated by a blank line or
!    a Control-C (^C).
!
! ENVIRONMENT:
!
!--

!+
! TABLE OF CONTENTS:
!-

FORWARD ROUTINE
    SORT_CHARS,                  ! Does sorting
    ALPHA_SORT;                  ! Main routine

!+
! INCLUDE FILES:
!-

LIBRARY 'CELPKG'; 1             ! Interface to cell
                                 ! package

%BLISS32( LIBRARY 'SYS$LIBRARY:STARLET';) 2

REQUIRE 'SYS$LIBRARY:TUTIO'; 3

!+
! EQUATED SYMBOLS:
!-

LITERAL
    TRUE=       1,
    FALSE=      0,
    LINE_BUFF_SIZE=  133; 4

!+
! OWN STORAGE:
!-

OWN

    HEADER:     HEADER_STR
        PRESET( [FLINK] = HEADER,
                [BLINK] = HEADER,
                [SIZEF] = HEADER_SIZE ), 5

    LINE_BUFF:  VECTOR[CH$ALLOCATION(LINE_BUFF_SIZE)]; 6
```

1. Use a LIBRARY declaration to include GET_CELL, REL_CELL, INS_CELL, REM_CELL, and the CELL data structure.
2. If BLISS–32, use a LIBRARY declaration to include the definition of SS$_NORMAL. This is used by the main routine as the return value.
3. Use a REQUIRE declaration to include the terminal I/O functions.

4 Limit the input line length to 132 and set the buffer size one character longer to detect longer lines.

5 Allocate and initialize a list header in static storage.

6 Allocate the line buffer.

Note

In all three declarations, the file type is allowed to have a default. This simplifies the writing of portable software, because the compiler searches for language-dependent extensions (such as .R32) before searching for .REQ.

9.1.2 Routine SORT_CHARS

```
ROUTINE SORT_CHARS (LEN, CH_PTR) =

!++
! FUNCTIONAL DESCRIPTION:
!
!     The input to this routine is a character sequence,
!     identified by LEN and CH_PTR. All non-uppercase
!     alphabetic characters are removed from the sequence
!     and the remaining characters are put in alphabetical
!     order. The resulting sequence is placed in the character
!     data segment originally occupied by the input character
!     sequence.
!
! FORMAL PARAMETERS:
!
!     LEN     - Length of character sequence.
!     CH_PTR  - Character pointer to the first
!               character in sequence.
!
! IMPLICIT INPUTS:
!
!     NONE
!
! ROUTINE VALUE:
!
!     Length of the resulting character sequence.
!
! SIDE EFFECTS:
!
!     The storage pointed to by CH_PTR is modified.
!
!--

    BEGIN
    LOCAL
        CHAR_PTR,                       ! A character pointer
        CURRENT_CELL:   REF CELL_STR;   ! A cell pointer

    CHAR_PTR = .CH_PTR; 1

    INCR I FROM 1 TO .LEN DO 2
        BEGIN
        LOCAL
            CHAR;                       ! A character

        CHAR = CH$RCHAR_A(CHAR_PTR);    ! Read character and
                                        ! advance pointer
```

```
    IF .CHAR GEQ %C'A' AND .CHAR LEQ %C'Z' 3
    THEN
        BEGIN
        LOCAL
            CELL:   REF CELL_STR;   ! A cell pointer.

        CURRENT_CELL = .HEADER[FLINK]; 4

        UNTIL .CURRENT_CELL EQLA HEADER DO 5

            BEGIN
            IF .CURRENT_CELL[VALUEF] GEQ .CHAR
            THEN
                EXITLOOP; 6
            CURRENT_CELL = .CURRENT_CELL[FLINK]
            END;

        CELL = GET_CELL(CELL_SIZE); 7
        CELL[VALUEF] = .CHAR;
        INS_CELL( .CELL, .CURRENT_CELL[BLINK] )
        END

    END;

CHAR_PTR = .CH_PTR; 8

UNTIL
    REM_CELL (CURRENT_CELL = .HEADER[FLINK]) EQL EMPTYK
DO 9

    BEGIN
    CH$WCHAR_A( .CURRENT_CELL[VALUEF], CHAR_PTR ); 10
    REL_CELL(.CURRENT_CELL)
    END;

CH$DIFF(.CHAR_PTR, .CH_PTR) 11
END;                                !End of SORT_CHARS
```

1 Initialize CHAR_PTR to point to first character in sequence.

2 In this loop, get a cell for each uppercase character in the sequence and insert the cell in a doubly linked list sorted by character values.

3 Test for uppercase characters.

4 Initialize CURRENT_CELL to the address of the list header.

5 Move forward through the list of cells until you reach the last. The last cell is recognized by its forward pointer, which points back to the list header.

   Note that the EQLA operator (although synonymous with EQL) is used to remind the programmer that addresses are being compared.

6 If the character value stored in this cell is greater than or equal to the last character value read, exit the loop. (You want to insert the new character before this one.)

   Note that if the loop runs to completion, and the EXITLOOP is never executed, CURRENT_CELL is left pointing to the list header. Inserting before the list header is equivalent to inserting at the end of the list (desired behavior).

7 Get a cell from heap storage, store the character value there, and insert the cell before the one pointed to by CURRENT_CELL.

8 Reset CHAR_PTR to point to the first character in the sequence.

9 For each cell in the doubly-linked list, write the character into the character sequence and delete the cell.

10 Write the character and advance the pointer.

11 Compute the length of the resulting character sequence by the difference between the pointer to the first character position and the next one to be written.

9.1.3 Routine ALPHA_SORT

```
ROUTINE ALPHA_SORT=

!++
! FUNCTIONAL DESCRIPTION:
!
!     Main routine of program. Loops until a null input line
!     is read from the terminal.
!
! FORMAL PARAMETERS:
!
!     NONE.
!
! IMPLICIT INPUTS:
!
!     NONE
!
! ROUTINE VALUE:
!
!     Return SS$_NORMAL to VMS.
!     Return 0 to other environments (value is ignored).
!
! SIDE EFFECTS:
!
!     NONE
!
!--

    BEGIN
    LABEL
        LOOP_BODY;

    WHILE TRUE DO
LOOP_BODY:
        BEGIN
        LOCAL
            LEN;                        ! Number of chars
                                        ! in line buffer

        TTY_PUT_QUO('INPUT:'); 1
        LEN = TTY_GET_LINE( LINE_BUFF, LINE_BUFF_SIZE );

        IF .LEN EQL 0 THEN EXITLOOP;    ! EOF exit.

        IF .LEN EQL LINE_BUFF_SIZE
        THEN
            BEGIN
            TTY_PUT_CRLF();
            TTY_PUT_QUO( 'Input line is too long.');
            TTY_PUT_CRLF();
            LEAVE LOOP_BODY             ! Exit block but
            END;                        ! continue looping.

        LEN = SORT_CHARS( .LEN, CH$PTR(LINE_BUFF) ); 2

        TTY_PUT_QUO('OUTPUT:'); 3
        TTY_PUT_MSG( LINE_BUFF, .LEN );
        TTY_PUT_CRLF()
        END;                            ! End of main loop.
```

```
    %IF %BLISS(BLISS32)
    %THEN
        SS$_NORMAL
    %ELSE
        0
    %FI
    END;                                ! End of ALPHA_SORT

END                                     ! End of module
ELUDOM
```

1 Prompt for input.

2 Sort characters.

3 Print output.

9.1.4 Cell-Handling Package

```
%TITLE 'Cell-Handling Package'

!++
! FACILITY:     CELL-HANDLING PACKAGE
!
! ABSTRACT:
!
!     This package contains macros to create and delete cells,
!     as well as to insert cells on and remove them from
!     doubly-linked lists.
!
!
! ENVIRONMENT:
!
!     The VAX Common RTL routines LIB$GET_VM and LIB$FREE_VM
!     are used to get and free heap storage.
!
!     The VAX-11 hardware queue instructions INSQUE and REMQUE
!     are used to insert and delete cells from lists.  See the
!     VAX Architecture Handbook for details.
!
!--

!+
! TABLE OF CONTENTS:
!     GET_CELL   Allocate a cell and return its address.
!     REL_CELL   Deallocate a cell.
!     INS_CELL   Insert a cell in a list.
!     REM_CELL   Remove a cell from a list and return a
!                completion status.
!-

!+
! EQUATED SYMBOLS:
!-

!+
!     Values returned by REM_CELL.
!
!     N.B. That the values are carefully chosen to be
!     identical with those returned by the VAX-11 REMQUE
!     builtin function.
!-
LITERAL
    EMPTYK=     3,      ! List was already empty
    LASTK=      2,      ! Last element removed, list now empty
    NOTLASTK=   0;      ! Element removed, but more remain
```

```
!+
! EXTERNAL REFERENCES:
!-

EXTERNAL ROUTINE 1
    LIB$GET_VM : ADDRESSING_MODE(GENERAL),
    LIB$FREE_VM : ADDRESSING_MODE(GENERAL);

!+
!  Cell Data Structure Definitions 2
!-

FIELD 3
    COMMON_FLDS=
        SET
        FLINK=  [0, 0, 32, 0],        ! Successor pointer
        BLINK=  [1, 0, 32, 0],        ! Predecessor pointer
        SIZEF=  [2, 0, 16, 0]         ! Cell size (in fullwords)
        TES;

FIELD
    VALUEF=     [2, 16, 8, 0];        ! Value part

LITERAL
    HEADER_SIZE=        3,            ! Size of a list-header
    CELL_SIZE=          3;            ! Size of a cell on the
                                      ! list

MACRO 4
    HEADER_STR= BLOCK[HEADER_SIZE] FIELD(COMMON_FLDS) %,
    CELL_STR=   BLOCK[CELL_SIZE] FIELD(COMMON_FLDS, VALUEF) %;
```

1. VMS Common RTL routines; GENERAL is required to get position independent references.
2. Defining a data structure using FIELD, LITERAL, and MACRO declarations is the recommended programming technique.
3. Declare names for the fields and sizes of cells.
4. Declare macros for statically allocating cells and map the appropriate structure on dynamically specified cells.

   Note that you can reference COMMON_FLDS (FLINK, BLINK, SIZEF) for things declared as list headers or cells; but you can reference the VALUEF only for cells.

9.1.5 Macro REL_CELL

```
MACRO REL_CELL( CELL ) =

!++
! FUNCTIONAL DESCRIPTION:
!
!     Deallocates storage for the cell at the specified
!     address, returning the storage to the heap.
!
! FORMAL PARAMETERS:
!
!     CELL    - Address of a cell.
!
!--

    BEGIN
    BIND
        C = (CELL) : CELL_STR; 1

    LIB$FREE_VM( %REF( .C[SIZEF] * %UPVAL ), %REF( C ) ) 2
```

```
    END %;                      ! End of REL_CELL
```

1 Declare C as a name bound to the address of a cell, and associate CELL_STR with C.

Note that the above is done for two reasons: First, to permit a structured-reference to the SIZEF field and second, to cause side-effects in the argument passed to REL_CELL to occur only once. For example, if the macro-formal was twice referenced and the macro invoked with REL_CELL (REM_CELL(.HEADER)), then REM_CELL would remove two cells, while REL_CELL released memory for only the second cell.

Note also that it is good programming practice to parenthesize macro formals.

2 Call the VMS Run-Time Library routine to release the appropriate number of bytes (addressable units) of storage.

Note that the arguments are passed by reference using %REF. The first argument is the address of a fullword containing the size (in addressable units) of the data segment to be released, while the second argument is the address of a fullword containing the address of the data segment to be released.

The size (in fullword units) is multiplied by %UPVAL (units per value) to convert to a size specified in addressable units.

9.1.6 Macro INS_CELL

```
MACRO INS_CELL( THIS_CELL, PREV_CELL ) =

!++
! FUNCTIONAL DESCRIPTION:
!
!     Insert the cell whose address is THIS_CELL in the
!     doubly-linked list, putting it after the cell whose
!     address is PREV_CELL.
!
! FORMAL PARAMETERS:
!
!     THIS_CELL       - Address of the cell to be inserted.
!     PREV_CELL       - Address of the cell after which
!                       THIS_CELL is to be inserted.
!
! SIDE EFFECTS:
!
!     Fields in THIS_CELL, PREV_CELL, and the successor to
!     PREV_CELL are modified.
!
!--

    BEGIN
    BUILTIN
        INSQUE;                 ! VAX-11 Insert Entry in
                                ! Queue Instruction.

    INSQUE( (THIS_CELL), (PREV_CELL) ); 1

    END %;                      ! End of INS_CELL
```

1 The semicolon is present here to cause the macro to be effectively NOVALUE. Any attempt to use this macro in a value-required context will cause the compiler to issue a diagnostic message.

9.1.7 Macro GET_CELL

```
MACRO GET_CELL( SIZE ) =

!++
! FUNCTIONAL DESCRIPTION:
!
!     Allocates storage for a cell of the specified size and
!     initializes the forward and backward pointers to point
!     to the cell.
!
! FORMAL PARAMETERS:
!
!     SIZE    - Size of cell (number of BLISS--values in cell).
!
! RETURN VALUE:
!
!     Address of the CELL allocated.
!
!--

    BEGIN
    LOCAL
        CELL:   REF CELL_STR;

    LIB$GET_VM( %REF((SIZE) * %UPVAL), CELL ); 1

    CELL[FLINK] = .CELL;
    CELL[BLINK] = .CELL;
    CELL[SIZEF] = (SIZE); 2

    .CELL
    END %;                              ! End of GET_CELL
```

1. Call the VMS Run-Time Library routine to allocate the appropriate number of bytes (addressable units) of storage.

   Note that as in REL_CELL, the arguments are passed by reference using %REF. The first argument is the address of a fullword containing the size (in addressable units) of the data segment to be allocated, while the second argument is the address of a fullword whose content is set to the address of the data segment allocated.

   The size (in fullword units) is multiplied by %UPVAL (units per value) to convert to a size specified in addressable units.

2. Note that if the macro formal SIZE was not parenthesized, the call GET_CELL(.X + 1) would result in a size computation of .X + 1 * %UPVAL. For BLISS–32, %UPVAL has a value of 4; thus the size requested would be .X + 4 and not (.X + 1) * 4.

9.1.8 Macro REM_CELL

```
MACRO REM_CELL( THIS_CELL ) =
```

```
!++
! FUNCTIONAL DESCRIPTION:
!
!     Remove the specified cell from its list. Its forward
!     and backward pointers are left pointing to the cell.
!
! FORMAL PARAMETERS:
!
!     THIS_CELL       - Address of the cell to be removed.
!
! COMPLETION CODES:
!
!     EMPTYK          - List was empty, no cell removed.
!     LASTK           - List now empty, last cell removed.
!     NOTLASTK        - List not empty, a cell was removed.
!
! SIDE EFFECTS:
!
!     Fields in THIS_CELL, its predecessor and successor are
!     modified.
!
!--

    BEGIN
    BUILTIN
        REMQUE;                 ! VAX-11 Remove Entry from
                                ! Queue Instruction.

    LOCAL
        TEMP;                   ! Temporary for entry removed
                                ! from QUEUE.

    REMQUE( (THIS_CELL), TEMP )

    END %;                      ! End of REM_CELL
```

9.2 Example 2: The CALC Program

This program accepts a simple integer expression for basic calculation to demonstrate the use of error recovery and parse/calculate routines. The program consists of seven routines and three INCLUDE files.

The CALC program uses the following facilities:

- STARLET—For system trap values
- TUTIO—Tutorial Terminal I/O Package
- CONDIT—Condition handling routine

The following steps are used to run the CALC program:

1. Compile the CALC program:

   ```
   $ BLISS CALC
   ```

2. Link the CALC program:

   ```
   $ LINK CALC
   ```

3. Run the CALC program:

```
$ RUN CALC
CALC: A hierarchical calculator.
Enter expression and terminate with
equal sign (=) or carriage return.
An empty line exits the program.

CALC>
```

4. Examples of responses to the input prompt follow:

```
CALC> 3*((2+4)/2)=RESULT: 9
CALC>
```

or:

```
CALC> 2*4<CR>
RESULT: 8
CALC> <CR>
$
```

9.2.1 Module CALCULATE

```
MODULE CALCULATE (MAIN = CALC_DRIVER) =

!+
!  FUNCTION
!
!      Demonstrate the use of the ENABLE function to do
!      error recovery by implementing a simple hierarchical
!      integer calculator that will use enable routines and
!      the UNWIND feature to get out of any nested routines
!      and prepare to parse/calculate the next expression.
!
!  ALLOWED SYNTAX
!
!      The standard binary functions (+,-,*,/) are supported,
!      as are parenthesized expressions. An equal sign signals
!      the end of the expression. Unary functions are, however,
!      not supported. All numbers and results are integers; no
!      real or floating point numbers are accepted.
!
!      Thus the following are legal expressions:
!
!               3*((3+2)/2)=
!               6=
!
!      and the following are not:
!
!               -3*4=
!               3.4*6=
!
!  EXIT CONDITIONS
!
!      A carriage return, CTRL/C, or a CTRL/Y terminates the
!      program.
!-
```

```
BEGIN
FORWARD ROUTINE
    CALC_DRIVER,        ! Unwinds to here on errors.
    MAIN : NOVALUE,     ! Establisher routine; results to terminal.
    READEXPRESSION,     ! Handles addition and subtraction.
    READTERM,           ! Handles multiplication and division.
    READFACTOR,         ! Dispatches based on digit or "(".
    READNUMBER,         ! Convert from ASCII digits to integer.
    READCHAR : NOVALUE, ! Reads characters from the terminal.
    CATCH_ERRORS;       ! Handler routine for all signals.

BUILTIN
    BICPSW,             ! Clear bits in the PSW.
    BISPSW;             ! Set bits in the PSW.

LIBRARY 'SYS$LIBRARY:STARLET';  1

SWITCHES LIST (NOSOURCE, NOOBJECT);
REQUIRE 'SYS$LIBRARY:TUTIO.R32';  2
REQUIRE 'CONDIT.R32';  3
SWITCHES LIST (SOURCE, OBJECT);

LITERAL
    END_OF_EXPRESSION = %C'=',
    CR = %O'15',
    TRUE = 1;

LITERAL  4
    INVALID_EXPRESSION =    STS$VALUE (CODE = 1),
    MISSING_RIGHT_PAREN =   STS$VALUE (CODE = 2),
    ILLEGAL_CHARACTER =     STS$VALUE (CODE = 3);

OWN
    CURCHAR : BYTE;  5

MACRO
    IN_DIGITS = .CURCHAR GEQ %C'0' AND .CURCHAR LEQ %C'9' %,
    IN_ADDING_OPS = .CURCHAR EQL %C'+' OR .CURCHAR EQL %C'-' %,
    IN_MUL_OPS = .CURCHAR EQL %C'*' OR .CURCHAR EQL %C'/' %;
```

1 STARLET is required for the PSL interrupt vector masks and to process system trap values.

2 TUTIO contains the TTY_xxx routines.

3 CONDIT is required to provide the condition handling macros.

4 The LITERALS are user defined signal codes.

5 CURCHAR is the current character being acted upon.

9.2.2 Routine CALC_DRIVER

```
ROUTINE CALC_DRIVER =
```

```
!+
!  FUNCTION
!
!      Prime the pump by reading in the first character.
!      Provide an enclosing routine that will be returned
!      to as the handler routine CATCH_ERRORS unwinds.
!
!  INPUTS
!
!      NONE.
!
!  OUTPUTS
!
!      NONE.
!
!  IMPLICIT INPUTS
!
!      CURCHAR -     The current character being parsed.
!
!  IMPLICIT OUTPUTS
!
!      NONE.
!
!  ROUTINE VALUE
!
!      SS$_NORMAL
!
!-
    BEGIN  1
    TTY_PUT_QUO ('CALC: A hierarchical calculator.');
    TTY_PUT_CRLF ();
    TTY_PUT_QUO ('Enter expression and terminate with');
    TTY_PUT_CRLF ();
    TTY_PUT_QUO (' equal sign (=) or carriage return.');
    TTY_PUT_CRLF ();
    TTY_PUT_QUO ('An empty line exits the program.');
    TTY_PUT_CRLF ();  TTY_PUT_CRLF ();
    TTY_PUT_QUO ('CALC> ');

    READCHAR ();

    WHILE .CURCHAR NEQ END_OF_EXPRESSION DO

        BEGIN
        MAIN ();
        TTY_PUT_QUO ('CALC> ');
        READCHAR ()
        END;

    RETURN SS$_NORMAL  2
    END;
```

1. This code provides the user interface to the program and calls the READCHAR routine for character input.
2. SS$_NORMAL returns normal status to the monitor.

9.2.3 Routine MAIN

```
ROUTINE MAIN : NOVALUE =

!+
!  FUNCTION
!
!      The main driver routine for actual expression
!      parsing, prints the value on the terminal if
!      a successful calculation takes place and sets
!      up the condition handler.
!
!  INPUTS
!
!      NONE.
!
!  OUTPUTS
!
!      NONE.
!
!  IMPLICIT INPUTS
!
!      CURCHAR -       The current character being parsed.
!
!  IMPLICIT OUTPUTS
!
!      NONE.
!
!  ROUTINE VALUE
!
!      NONE
!-
    BEGIN
    LOCAL
        RESULT;

    ENABLE
        CATCH_ERRORS;  1

        RESULT = READEXPRESSION ();

        IF .CURCHAR EQL END_OF_EXPRESSION
        THEN  2
            BEGIN
            TTY_PUT_QUO ('RESULT: ');
            TTY_PUT_INTEGER (.RESULT, 10, 1);
            TTY_PUT_CRLF ()
            END
        ELSE
            SIGNAL (INVALID_EXPRESSION)
    END;
```

1 ENABLE the CATCH_ERRORS routine to handle all errors.

2 This code displays the result of the calculation on the screen.

9.2.4 Routine READEXPRESSION

```
ROUTINE READEXPRESSION =

!+
!  FUNCTION
!
!      Do the addition and subtraction Terms. The bit in
!      the Program Status Longword (that allows the user
!      to get control during integer overflow) is set before
!      the actual addition and cleared immediately after;
!      if overflow occurs here, the program knows what to do
!      about it.
!  INPUTS
!
!      NONE.
!
!  OUTPUTS
!
!      NONE.
!
!  IMPLICIT INPUTS
!
!       CURCHAR -     The current character being parsed.
!
!  IMPLICIT OUTPUTS
!
!      NONE.
!
!  ROUTINE VALUE
!
!      The result of an addition to or subtraction from
!      the current value of the expression.
!-
    BEGIN
    LOCAL
        EXPRVALUE;

    EXPRVALUE = READTERM ();

    WHILE IN_ADDING_OPS DO
        BEGIN
        LOCAL
            ADDOP;

        ADDOP = .CURCHAR;
        READCHAR ();
        BISPSW (%REF (PSL$M_IV));  1

        IF .ADDOP EQL %C'+'
        THEN
            EXPRVALUE = .EXPRVALUE + READTERM ()
        ELSE
            EXPRVALUE = .EXPRVALUE - READTERM ();
        BICPSW (%REF (PSL$M_IV))  2
        END;

    .EXPRVALUE
    END;
```

1 Turn on integer overflow.

2 Turn off integer overflow.

9.2.5 Routine READTERM

```
ROUTINE READTERM =

!+
!  FUNCTION
!
!      Do multiplication and division. Turn user control of
!      overflow on and off as required. Division by zero is
!      detected by the hardware. The enable mechanism is
!      called and routine CATCH_ERRORS (the handler routine
!      for all errors) takes control. The proper message is
!      displayed and unwinding occurs back to the outermost
!      routine (CALC_DRIVER).
!
!  INPUTS
!
!      NONE.
!
!  OUTPUTS
!
!      NONE.
!
!  IMPLICIT INPUTS
!
!       CURCHAR -     The current character being parsed.
!
!  IMPLICIT OUTPUTS
!
!      NONE.
!
!  ROUTINE VALUE
!
!       Value of the expression calculated through
!       multiplication or division.
!-

    BEGIN
    LOCAL
        TERM_VALUE;

    TERM_VALUE = READFACTOR ();

    WHILE IN_MUL_OPS DO
        BEGIN
        LOCAL
            MULOP : BYTE;

        MULOP = .CURCHAR;
        READCHAR ();

        IF .MULOP EQL %C'*'
        THEN
            BEGIN
            BISPSW (%REF (PSL$M_IV));
            TERM_VALUE = .TERM_VALUE * READFACTOR ();
            BICPSW (%REF (PSL$M_IV))
            END
        ELSE
            TERM_VALUE = .TERM_VALUE / READFACTOR ()
        END;
    .TERM_VALUE
    END;
```

9.2.6 Routine READFACTOR

```
ROUTINE READFACTOR =

!+
!  FUNCTION
!
!      If the current character is an ASCII representation
!      of a digit, then call READNUMBER to get the value
!      of the number. If the character is a "(", call
!      READEXPRESSION and return the result (The value of
!      the parenthesized expression).
!
!  INPUTS
!
!      NONE.
!
!  OUTPUTS
!
!      NONE.
!
!  IMPLICIT INPUTS
!
!      CURCHAR -     The current character being parsed.
!
!  IMPLICIT OUTPUTS
!
!      NONE.
!
!  ROUTINE VALUE
!
!      Value of a number or a parenthesized expression.
!-

    BEGIN
    SELECTONE .CURCHAR OF
        SET
        [%C'0' TO %C'9']:
            RETURN READNUMBER (); 1

        [%C'(']:
            BEGIN 2
            LOCAL
                FACTOR_VALUE;
            READCHAR ();

            FACTOR_VALUE = READEXPRESSION (); 3
            IF .CURCHAR EQL %C')'
            THEN  4
                BEGIN
                READCHAR ();
                RETURN .FACTOR_VALUE
                END
            ELSE  5
                SIGNAL (MISSING_RIGHT_PAREN)
            END;

        [OTHERWISE]:
            SIGNAL (illegal_character) 6
        TES
    END;
```

1. At this point the routine is parsing a number.
2. The number is a parenthesized expression.
3. Determine the value of the enclosed expression.

4 Determine if the expression is legal.

5 Determine if the expression has matching parenthesis.

6 Issue a message if something unexpected has been entered.

9.2.7 Routine READNUMBER

```
ROUTINE READNUMBER =

!+
!  FUNCTION
!
!      Convert a string of ASCII digits to an integer number.
!
!  INPUTS
!
!      NONE.
!
!  OUTPUTS
!
!      NONE.
!
!  IMPLICIT INPUTS
!
!      CURCHAR -     The current character being parsed.
!
!  IMPLICIT OUTPUTS
!
!      NONE.
!
!  ROUTINE VALUE
!
!      Integer value of a string of ASCII digits.
!
!-
    BEGIN
    LOCAL
        NUM_VALUE : initial(0);

    WHILE IN_DIGITS DO
        BEGIN  1
        LOCAL
            DIGIT : BYTE;
        DIGIT = .CURCHAR - %C'0';
        BISPSW (%REF (PSL$M_IV));  2
        NUM_VALUE = .NUM_VALUE * 10 + .DIGIT;
        BICPSW (%REF (PSL$M_IV));
        READCHAR ()
        END;
    .NUM_VALUE
    END;
```

1 Convert a string of ASCII digits to an integer number.

2 Check for too big a number.

9.2.8 Routine READCHAR

```
ROUTINE READCHAR : NOVALUE =

!+
!  FUNCTION
!
!      Read input from terminal until a non-blank character is
!      found. If a carriage return is found, call that an
!      end-of-expression.
!
!  INPUTS
!
!      NONE.
!
!  OUTPUTS
!
!      NONE.
!
!  IMPLICIT INPUTS
!
!      NONE.
!
!  IMPLICIT OUTPUTS
!
!      CURCHAR -       The current character to be parsed.
!
!  ROUTINE VALUE
!
!      NONE.
!
!-

    DO
        BEGIN
        CURCHAR = TTY_GET_CHAR ();

        IF .CURCHAR EQL CR
        THEN
            CURCHAR = END_OF_EXPRESSION  1
        END
    UNTIL .CURCHAR NEQ %C' ';  2
```

1 End of line is also END_OF_EXPRESSION.

2 Loop until a nonblank character is encountered.

9.2.9 Routine CATCH_ERRORS

```
ROUTINE CATCH_ERRORS (sig: REF VECTOR, mech, enab) =
```

```
!+
!  FUNCTION
!
!      This is the condition handler for all expected errors.
!      Division by zero is detected by the hardware and if the
!      program has control (via the proper bit set in the PSL),
!      Integer Overflow is also detected. Otherwise, the program
!      traps out without enabling condition handling. Explicit
!      SIGNAL calls occur for all other detected conditions and
!      an appropriate message is displayed on the terminal.
!
!      After the user has been notified, an UNWIND is signalled.
!      The establisher routine is the main driver; returning
!      to its caller sets up the parsing for a completely
!      new expression.
!
!      The rationale for having all detected errors unwind is
!      that routines can be very busy calling each other, and
!      "fixing things up" on the fly is unreasonable. Also,
!      once an error occurs, the value of the expression that
!      caused the error is worthless; thus, continued input for
!      the expression is refused and the program prepares for
!      the next expression.
!
!  INPUTS
!
!       SIG -        Signal vector
!       MECH -       Mechanism vector
!       ENBL -       Enable vector
!
!  OUTPUTS
!
!      NONE.
!
!  IMPLICIT INPUTS
!
!      NONE.
!
!  IMPLICIT OUTPUTS
!
!      NONE.
!
!  ROUTINE VALUE
!
!      NONE.
!
!-
    BEGIN
    BIND
        COND = SIG [1] : CONDITION_VALUE;

    SELECTONE TRUE OF  1
        SET
        [STS$MATCH (.COND, INVALID_EXPRESSION)]:
            TTY_PUT_QUO ('      INVALID EXPRESSION');

        [STS$MATCH (.COND, MISSING_RIGHT_PAREN)]:
            TTY_PUT_QUO ('      Missing a right parenthesis');

        [STS$MATCH (.COND, ILLEGAL_CHARACTER)]:
            TTY_PUT_QUO ('      Parsed an illegal character');

        [STS$MATCH (.COND, SS$_INTDIV)]:
            TTY_PUT_QUO ('      Attempted division by zero');
```

```
        [STS$MATCH (.COND, SS$_INTOVF)]:
            TTY_PUT_QUO ('      Integer overflow');

        [OTHERWISE]:  2
            RETURN 0
        TES;

    TTY_PUT_QUO ('; calculation stopped.');
    TTY_PUT_CRLF ();

    SETUNWIND ();  3
    0  4
    END;
END ELUDOM
```

1 Determine action to be taken based on the type of error encountered.

2 Error is possibly unexpected or an UNWIND error; in either case do nothing.

3 Get out of the recursion and prepare to parse the next expression.

4 Request normal UNWIND.

A
Summary of Command Syntax

This appendix summarizes BLISS command syntax, qualifier defaults, and abbreviations.

A.1 Command-Line Syntax

compilation-request	$ BLISS [/qualifier . . .] input-spec, . . .
input-spec	file-spec[+ . . .] [/qualifier . . .]

```
              { check qualifier             }
              { general qualifiers          }
              { machine-code-list qualifier }
              { optimization qualifier      }
qualifier     { output qualifiers           }
              { reference qualifier         }
              { source-list qualifier       }
              { terminal qualifier          }
```

A.2 File Specification Summary

file-spec	{ node:: } { dev: } { [dir] } file { .type } { ;ver }
node	1- to 6-character alphanumeric network node name and optionally a 0- to 42-character access control string
dev	Any logical or physical device name; includes device type, controller designation, and unit number
dir	Any valid directory or subdirectory name
file	1 to 39 alphanumeric characters, including underscore (_), hyphen (-), and dollar sign ($)
type	0 to 39 alphanumeric characters preceded by a period
ver	Version number from 1 to 32767

A.3 Qualifier Syntax

```
check qualfier      /CHECK = { (check-value , . . . ) }
                             { check-value            }

                 { FIELD        | NOFIELD       }
check-value      { INITIAL      | NOINITIAL     }
                 { OPTIMIZE     | NOOPTIMIZE    }
                 { REDECLARE    | NOREDECLARE   }
```

general qualifiers	/CODE	/NOCODE
	/DEBUG	/NODEBUG
	/ERROR_LIMIT {:=value}	
	/TRACEBACK	/NOTRACEBACK
	/VARIANT {:value}	

machine-code-list qualifier

/MACHINE_CODE_LIST= { (code-value , . . .) | code-value }

code-value	OBJECT	NOOBJECT
	ASSEMBLER	NOASSEMBLER
	BINARY	NOBINARY
	COMMENTARY	NOCOMMENTARY
	SYMBOLIC	NOSYMBOLIC
	UNIQUE_NAMES	NOUNIQUE_NAMES

optimize qualifier

/OPTIMIZE= { (optimize-value , . . .) | optimize-value }

optimize-value	LEVEL : optimize-level	
	QUICK	NOQUICK
	SAFE	NOSAFE
	SPEED	SPACE

optimize-level

{ 0 | 1 | 2 | 3 }

output qualifiers	/ANALYSIS_DATA {=file-spec}	/NOANALYSIS_DATA
	/LIST {=file-spec}	/NOLIST
	/LIBRARY {=file-spec}	/NOLIBRARY
	/OBJECT {=file-spec}	/NOOBJECT

source-list qualifier

/SOURCE_LIST= { (source-value , . . .) | source-value }

source-value	EXPAND_MACROS	NOEXPAND_MACROS
	HEADER	NOHEADER
	LIBRARY	NOLIBRARY
	PAGE_SIZE : number of lines	
	REQUIRE	NOREQUIRE
	SOURCE	NOSOURCE
	TRACE_MACROS	NOTRACE_MACROS

number of lines

{ 20 | 21 | 22 | . . . }

reference qualifier — { /CROSS_REFERENCE { = { reference-value } } }

reference-value — { MULTIPLE | NOMULTIPLE }

terminal qualifier — /TERMINAL= { (terminal-value , . . .) | terminal-value }

terminal-value — { ERRORS | NOERRORS ; STATISTICS | NOSTATISTICS }

A.4 Qualifier Defaults

The following qualifiers are assumed by default:

```
/CHECK=(FIELD,INITIAL,OPTIMIZE,NOREDECLARE)
/CODE
/ERROR_LIMIT=30
/LIST (in batch mode)
/MACHINE_CODE_LIST=(NOASSEMBLER,SYMBOLIC,BINARY,
            OBJECT COMMENTARY,
            NOUNIQUE_NAMES)
/NOANALYSIS_DATA
/NOCROSS_REFERENCE or /CROSS_REFERENCE(NOMULTIPLE)
/NODEBUG
/NOLIBRARY
/NOLIST (in interactive mode)
/OBJECT=input-file-name.OBJ
/OPTIMIZE=(NOQUICK,SPACE,LEVEL:2,SAFE)
/SOURCE_LIST=(HEADER,PAGE_SIZE:58,NOLIBRARY,
        NOREQUIRE, NOEXPAND_MACROS,
        NOTRACE_MACROS,SOURCE)
/TERMINAL=(ERRORS,NOSTATISTICS)
/TRACEBACK
/VARIANT:0
```

A.5 Abbreviations

The abbreviations for the positive forms of the qualifiers and qualifier values are given below:

Qualifier	Value	Abbreviation
/ANALYSIS_DATA		/A
/CHECK		/CH
	FIELD	F
	INITIAL	I
	OPTIMIZE	O
	REDECLARE	R
/CODE		/C

Qualifier	Value	Abbreviation
/CROSS_REFERENCE		/CR
	MULTIPLE	M
/DEBUG		/D
/ERROR_LIMIT		/E
/LIBRARY		/LIB
/LIST		/LIS
/MACHINE_CODE_LIST		/M
	ASSEMBLER	A
	BINARY	B
	COMMENTARY	C
	OBJECT	O
	SYMBOLIC	S
	UNIQUE_NAMES	U
/OBJECT		/OB
/OPTIMIZE		/OP
	LEVEL	L
	QUICK	Q
	SAFE	SA
	SPACE	SPA
	SPEED	SPE
/SOURCE_LIST		/S
	EXPAND_MACROS	E
	HEADER	H
	LIBRARY	L
	PAGE_SIZE	P
	REQUIRE	R
	SOURCE	S
	TRACE_MACROS	T
/TERMINAL		/TE
	ERRORS	E
	STATISTICS	S
/TRACEBACK		/TR
/VARIANT		/V

For the negative form of a qualifier or value (where applicable), its positive-form abbreviation can be prefixed by “NO”.

B

Summary of Formatting Rules

The basic rule of indentation is that a block is indented one logical tab deeper than the current indentation level (one logical tab equals four spaces; two logical tabs equal one physical tab). The declarations and expressions of a block must be indented to the same level as the BEGIN-END delimiters.

The format for a declaration is as follows:

```
declaration-keyword
    declaration-item,           !comment
          .
          .
          .
    declaration-item;           !comment
```

where the declaration-keyword starts at the current indentation level and each declaration-item is further indented one logical tab.

Expressions generally have two formats: one for expressions that fit on one line and one for expressions that are longer. If the expression does not fit on one line, then keywords must appear on separate lines from subparts and subparts, are indented one tab. For example, IF expressions can be written in either of two formats:

```
IF test THEN consequence ELSE alternative;
```

or

```
IF test
THEN
    consequence
ELSE
    alternative;
```

The examples used in Chapter 2 are indented correctly, although all comments have been omitted in order to save space.

C
Module Template

This appendix contains a listing of the file MODULE.BLI, which is the standard template for BLISS modules and routines. A module has four parts: a preface, a declarative part, an executable part, and a closing part.

The module's preface (Section C.1) appears first. It provides documentation explaining the module's function, use, and history.

The module's declarative part appears next (Section C.2). This section provides a table of contents for the module (FORWARD ROUTINE declarations) and declarations of macros, equated symbols, OWN storage, externals, and so on.

The module's executable part (Section C.3), consisting of zero or more routines, comes next. The template for a routine is in this section. Note that a routine has three parts: a preface, a declarative part, and code.

Finally, every module has a closing part (Section C.4), which completes the syntax of a module.

The module template can be used either as a checklist for module organization and content or as the starting point in creating a new module.

The file MODULE.BLI is supplied as part of the BLISS support package, on logical device SYS$LIBRARY.

C.1 Module Preface

```
MODULE TEMPLATE (                         !
                IDENT = ' '
                                          ) =
BEGIN
!
!                                COPYRIGHT (C) 1982 BY
!                    DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
!
! THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND  COPIED
! ONLY  IN  ACCORDANCE  WITH  THE  TERMS  OF  SUCH  LICENSE AND WITH THE
! INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR  ANY  OTHER
! COPIES  THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
! OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE  IS  HEREBY
! TRANSFERRED.
!
! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE  WITHOUT  NOTICE
! AND  SHOULD  NOT  BE  CONSTRUED  AS  A COMMITMENT BY DIGITAL EQUIPMENT
! CORPORATION.
!
! DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR  RELIABILITY  OF  ITS
! SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
!
!++
! FACILITY:
!
! ABSTRACT:
```

```
!
!
! ENVIRONMENT:
!
! AUTHOR:      , CREATION DATE:
!
! MODIFIED BY:
!
!       , : VERSION
! 01    -
!--
```

C.2 Declarative Part of Module

```
!
! TABLE OF CONTENTS:
!

FORWARD ROUTINE
               ;                          !

!
! INCLUDE FILES:
!

!
! MACROS:
!

!
! EQUATED SYMBOLS:
!

!
! OWN STORAGE:
!

!
! EXTERNAL REFERENCES:
!

EXTERNAL ROUTINE
          ;                          !
```

C.3 Executable Part of Module

```
ROUTINE TEMP_EXAMPLE () :NOVALUE =     !
!++
! FUNCTIONAL DESCRIPTION:
!
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--
    BEGIN

    LOCAL
                ;                          !
    END;                                   !END OF TEMP_EXAMPLE
```

C.4 Closing Format

```
END                                        !END OF MODULE
ELUDOM
```

D
Implementation Limits

Each BLISS–32 compiler implementation has value limitations on the use of certain language constructs or system interfaces. These values are subject to change if experience indicates they are unsuitable.

D.1 BLISS–32 Constructs

Certain BLISS–32 constructs have the following maximum limits on their size and number:

Nested blocks containing declarations	64
Characters in a quoted string	1000
Actual parameters in a routine call	64
Structure formal parameters	31
Field components	32
Parameters of a FIELD attribute	128
Bytes initialized by a single PLIT (that is, the maximum byte count of a single PLIT)	65,535
Length of a character sequence (CH$ function argument)	65,535

D.2 System Interfaces

System interfaces have the following maximum limits:

Characters in an input source line	132
Simultaneously active (depth of nested) REQUIRE files	9

E

Error Messages

Whether an error is fatal to the creation of an object module (ERR) or a warning (WARN) is context dependent. Informational messages (INFO) have no effect on compilation. BLISS–32 creates an object module for a program that has warnings but no errors. However, such a program may fail to link or may fail to execute in the intended manner. In some of these error messages, the compiler provides variable information that points to the possible source of error. (See the example in Section 2.1 for an illustration.) In this appendix, any information enclosed in angle brackets (< >) describes the type of such variable information given. Fatal error messages appear at the end of the appendix.

000 Undeclared name: <name>

Explanation: The name shown has not previously been declared.

User Action: Declare the name.

001 Declaration following an expression in a block

Explanation: Declarations must precede expressions within a block.

User Action: Reinsert the declaration properly or create a new block.

002 Superfluous operand preceding "<operator-name>"

Explanation: An excess or unnecessary left operand precedes the operator named.

User Action: Remove the extra or unnecessary left operand.

003 BEGIN paired with right parenthesis

Explanation: A closed parenthesis has been encountered when the compiler expected an END.

User Action: Provide the appropriate pairing or insert a missing END keyword.

004 Missing operand preceding "<operator-name>"

Explanation: Required left operand is missing from infix operator named.

User Action: Insert missing left operand.

005 Control expression must be parenthesized

Explanation: Parenthesis is required to achieve intended result.

User Action: Insert missing parenthesis.

006 Superfluous operand following "<operator-name>"

Explanation: An extra or unecessary right operand follows the operator named.

User Action: Remove the excess or unecessary right operand.

007 Missing operand following “<operator-name>”

Explanation: Required right operand is missing from operator named.

User Action: Insert missing right operand.

008 Missing THEN following IF

Explanation: Conditional expression is incomplete.

User Action: Insert required keyword THEN.

009 Missing DO following WHILE or UNTIL

Explanation: Pre-tested loop expression is incomplete.

User Action: Insert required keyword DO.

010 Missing WHILE or UNTIL following DO

Explanation: Post-tested loop expression is incomplete.

User Action: Insert required keyword WHILE or UNTIL.

011 Name longer than 31 characters

Explanation: Maximum name length has been exceeded.

User Action: Reduce name length to 31 characters or less.

012 Missing DO following INCR or DECR

Explanation: Indexed loop expression is incomplete.

User Action: Insert required keyword DO.

013 Missing comma or right parenthesis in routine actual parameter list

Explanation: Each actual parameter in a list must be separated by a comma and the list must be ended by a close parenthesis.

User Action: Insert commas, as necessary, and/or close parenthesis.

014 Missing FROM following CASE

Explanation: Case expression is incomplete.

User Action: Insert required keyword FROM.

015 Missing TO following FROM in CASE expression

Explanation: Case expression is incomplete.

User Action: Insert required keyword TO.

016 Missing OF following TO in CASE expression

Explanation: Case expression is incomplete.

User Action: Insert required keyword OF.

017 Missing OF following SELECT

Explanation: Select expression is incomplete.

User Action: Insert required keyword OF.

018 Missing SET following OF in SELECT expression

Explanation: Select expression is incomplete.

User Action: Insert required keyword SET.

019 Missing colon following right bracket in SELECT expression

Explanation: Select line of select expression is incomplete.

User Action: Insert colon between select label expression's close bracket and select action expression.

020 Missing semicolon or TES following a SELECT action

Explanation: Select line of select expression is incomplete.

User Action: Insert required semicolon or keyword TES following select action expression.

021 Address arithmetic involving REGISTER variable <variable-name>

Explanation: An attempt has been made to use the value of a register name in an expression.

User Action: Correct the expression.

022 Field reference used as an expression has no value

Explanation: The reference is invalid as a fetch or assignment expression and cannot produce a value.

User Action: Evaluate and validate the expression.

023 Missing comma or right angle bracket in a field selector

Explanation: Field selector is incomplete.

User Action: Insert missing commas or close bracket.

024 Value in field selector outside permitted range

Explanation: The value has exceeded the field size or machine word boundaries of the dialect.

User Action: Correct the address, position, size, or sign expression value according to dialect restrictions.

025 Value of attribute outside permitted range

Explanation: The value used is larger than the legal range permits, such as UNSIGNED(37).

User Action: Correct the attribute value.

026 ALIGN request negative or exceeds that of PSECT (or stack)

Explanation: The alignment attribute boundary must be a positive integer that does not exceed the psect alignment boundary.

User Action: Correct the boundary value.

027 Illegal character in source text

Explanation: One of the 30 illegal nonprinting ASCII characters has been used (as other than data) in a BLISS module.

User Action: Only four nonprinting characters (blank, tab, vertical tab, and form feed) can be used.

028 Illegal parameter in call to lexical function <lexical-function-name>

Explanation: A parameter used with the named lexical function is invalid.

User Action: Check and correct parameter usage according to the definition of the function.

029 Attribute illegal in this declaration

Explanation: Attributes are restricted in use to certain declarations.

User Action: Remove the illegal attribute from the declaration.

030 Access formals must not appear in structure size-expression

Explanation: An access formal provides variable access to elements of a structure and should not be included with the expression defining structure size.

User Action: Remove the access formal from the structure size expression.

031 Conflicting or multiple specified attributes

Explanation: Contradictory or superfluous attributes have been used.

User Action: Check attribute usage in regard to specific definitions.

032 Two consecutive field selectors

Explanation: Irrational use of field selector portion of field reference.

User Action: Remove extra field selector or insert parenthesis to create a complete field reference, such as: .(.x<0,16>)<0,8>.

033 Syntax error in attribute

Explanation: An error has occurred in the coding of an attribute.

User Action: Correct the error using the appropriate syntax.

034 INITIAL value <integer> too large for field

Explanation: The integer value shown is too large for the designated field.

User Action: Decrease the value or increase the allocation unit.

035 The <attribute-name> attribute contradicts corresponding FORWARD declaration

Explanation: The attributes of a name in an own, global, or routine declaration must be identical to those used in the associated forward declaration.

User Action: Correct the syntax of the attribute named.

036 Literal value cannot be represented in the declared number of bits

Explanation: The literal value of a literal declaration is larger than the field specified by the storage attribute.

User Action: Check sign or bit count of range attribute.

037 Lower bound of a range exceeds upper bound

Explanation: The low bound value of the range of a case expression must not exceed the high bound value.

User Action: Correct the low bound value.

038 Number of routine actual parameters exceeds implementation limit of 64

Explanation: The number of input actual parameters for a routine declaration must not exceed 64.

User Action: Decrease the number of parameters to 64.

039 Name used in an expression has no value: <name>

Explanation: A name that cannot denote an arithmetic value has been used in an expression.

User Action: Correct the expression.

040 LEAVE not within the block labelled by <label-name>

Explanation: The leave expression is not within the block of the label named.

User Action: Insert the expression in the appropriate block.

041 Missing comma or right parenthesis in parameter list to lexical function <lexical-function-name>

Explanation: Each lexical actual parameter in a list must be separated by a comma and the list must be ended by a close parenthesis.

User Action: Insert the missing commas or close parenthesis.

042 Missing label name following LEAVE

Explanation: The leave expression is incomplete.

User Action: Insert the appropriate label name following the keyword LEAVE.

043 Label <label-name> already labels another block

Explanation: The label name shown has been declared for another labeled block.

User Action: Change the name of one block or the other.

044 EXITLOOP not within a loop

Explanation: An exit loop expression has been incorrectly used.

User Action: Insert the expression within the (innermost) loop to be exited.

045 Missing structure name following REF

Explanation: The structure attribute using keyword REF is incomplete.

User Action: Insert the missing structure name following the keyword.

046 Register <register-number> cannot be reserved

Explanation: The register defined by the number shown is not locally usable.

User Action: Specify another register.

047 Module prematurely ended by extra close bracket or missing open bracket

Explanation: The number of close brackets in a module must equal the number of open brackets.

User Action: Remove the extra right bracket or add the missing left bracket.

048 Syntax error in module head

Explanation: The module head is incorrectly coded.

User Action: Correct the module name or the syntax of the module switch list.

049 Invalid switch specified

Explanation: An invalid switches declaration has been used with the dialect.

User Action: Correct the use of the switches declaration.

050 Name already declared in this block: <name>

Explanation: The name shown has been declared more than once in the same block.

User Action: Remove all but one of the declarations within the block.

051 Syntax error in switch specification

Explanation: An error has occurred in the specification of the module switches or switches declaration.

User Action: Correct the switches.

052 Expression must be a compile-time constant

Explanation: The compiler requires a compile-time constant expression, and the expression used does not meet the criteria.

User Action: Evaluate and correct the expression.

053 Invalid attribute in declaration

Explanation: An illegal attribute has been used in the declaration.

User Action: Check the legality of the attributes used with the declaration.

054 Name in attribute list not declared as a structure or linkage name: <name>

Explanation: The name shown has not been used as a structure or linkage name in a structure or linkage declaration.

User Action: Correct or declare the name appropriately.

055 Missing equal sign in BIND or LITERAL declaration

Explanation: The name and value of a literal, bind data, or bind routine item must be separated by an equal sign.

User Action: Insert the missing equal sign.

056 Missing comma or semicolon following a declaration

Explanation: Each declaration in a list must be separated by a comma, and the last must be followed by a semicolon.

User Action: Insert the missing commas or semicolon.

057 Value of structure size-expression for REGISTER must not exceed 4

Explanation: Structure size expression exceeds maximum allowed value.

User Action: Correct the value of the register structure size expression.

058 Left parenthesis paired with END

Explanation: A pair of parentheses must be used to replace an END pair.

User Action: Provide the appropriate pairing or insert a missing BEGIN keyword.

059 Register <register-number> cannot be specifically declared

Explanation: Register number shown is beyond the allowable range of the dialect or is illegally declared, such as REGISTER R = 50.

User Action: Insert a valid register number.

060 Missing SET following OF in CASE expression

Explanation: Case expression is incomplete.

User Action: Insert required keyword SET.

061 Missing left bracket preceding a CASE- or SELECT-label

Explanation: Case or select expression is incomplete.

User Action: Insert missing open bracket.

062 MODULE declaration inside module body

Explanation: A module body cannot contain a module declaration.

User Action: Correct the module code.

063 More than one CASE-label matching the same CASE-index

Explanation: Only one case label value can match a given case index value.

User Action: Correct either the label or index value.

064 Value in CASE-label outside the range given by FROM and TO

Explanation: Value of case label is not within the range specified.

User Action: Correct the case label or range values.

065 Missing equal sign in ROUTINE declaration

Explanation: An equal sign must precede the routine body in a routine declaration.

User Action: Insert the missing equal sign.

066 Two consecutive operands with no intervening operator

Explanation: Operator expression is incomplete or illegal.

User Action: The compiler will usually insert an appropriate operator and continue (if not correct the expression).

067 Missing comma or right bracket following a CASE- or SELECT-label

Explanation: Each label in a list, for a case or select expression, must be separated by a comma and the list ended with a close bracket.

User Action: Insert missing commas or close bracket.

068 Name to be declared is a reserved word: <name>

Explanation: Reserved words cannot be declared by the user.

User Action: Select another name for the declaration.

069 Size-expression required in STRUCTURE declaration when storage is to be allocated

Explanation: When a structure is associated with a name in a data declaration an expression must be used to specify the amount of storage allocated.

User Action: Insert structure size expression.

070 Number of structure formal parameters exceeds implementation limit of 31

Explanation: Number of access formal parameters exceeds maximum allowed.

User Action: Reduce the number of parameters.

071 Missing comma or closing bracket in formal parameter list for <routine-or-macro-name>

Explanation: Each formal parameter in a list must be separated by a comma and the list ended with a right bracket.

User Action: Insert missing commas or right bracket.

072 Missing control variable name in INCR or DECR expression

Explanation: Indexed loop expression is incomplete.

User Action: Insert missing loop index name.

073 Missing equal sign in STRUCTURE or MACRO declaration

Explanation: An equal sign must precede the structure size expression or structure body or the macro body.

User Action: Insert the missing equal sign.

074 Missing actual parameter list for macro <macro-name>

Explanation: The actual parameters are missing from the macro call associated with the macro named.

User Action: Insert actual parameters to correspond with the formal name parameters from the declaration.

075 Missing closing bracket or unbalanced brackets in actual parameter list for macro <macro-name>

Explanation: There must be a right bracket for every left bracket used in the actual parameter list.

User Action: Correct the pairing of the open and close brackets.

076 Extra actual parameters for structure <name> referencing data segment <name>

Explanation: Superfluous access actual parameters in structure reference for structure and data segment named.

User Action: Correct the structure reference code.

077 Missing colon following right bracket in CASE expression

Explanation: Case expression is incomplete.

User Action: Insert colon following close bracket.

078 Name to be mapped is undeclared or not mappable: <name>

Explanation: Name shown is undeclared or does not lie within the scope of a data or data bind declaration of the same name.

User Action: Declare name or correct declaration in an appropriate manner.

079 Missing comma or right bracket in structure actual parameter list

Explanation: A comma must separate each access actual parameter in a list and the list must be ended with a close bracket.

User Action: Insert missing commas or close bracket.

080 Illegal characters in quoted string parameter of <lexical-function-name>

Explanation: The only valid ASCII characters for a quoted string are blanks, tabs, paired single quotes, and any printing character except an apostrophe.

User Action: Remove illegal characters, or use %STRING for all characters with %CHAR inserted before illegal ones.

081 Quoted string not terminated before end of line

Explanation: A quoted string character sequence extends over a line.

User Action: Using %STRING and open parenthesis, quote first character sequence and before end-of-line conclude with a comma; do the same with subsequent sequences and then conclude the last line with a close parenthesis.

082 Missing comma or right parenthesis following a PLIT, INITIAL or PRESET item

Explanation: Each item in the list must be separated by a comma and the list must be ended with close parenthesis.

User Action: Insert any missing comma or close parenthesis.

083 Actual parameter list for macro <macro-name> not terminated before end of program

Explanation: The actual parameter list in the call for the macro named must be ended by a close parenthesis or close bracket (right square or right angle) even if the list is empty.

User Action: Insert the missing close parenthesis or bracket.

084 Expression must be a link-time constant

Explanation: The compiler requires a link time constant expression, and the expression used does not meet the criteria.

User Action: Evaluate and correct the expression.

085 String literal too long for use outside a PLIT

Explanation: The numeric value of a string literal exceeds the word length for the dialect.

User Action: Reduce the length of the string or use a PLIT declaration.

086 Name declared FORWARD is not defined: <name>

Explanation: A name declared in a forward declaration must also be declared by an own or global declaration within the same block.

User Action: Make the proper declarations.

087 Size of initial value (<integer-value>) exceeds declared size (<integer-value>)

Explanation: The initial value shown is greater than the memory space reserved for it.

User Action: Decrease the initial value and/or increase the declared size value.

088 Missing quoted string following <lexical-function-name>

Explanation: The lexical function shown requires a quoted string.

User Action: Insert the required quoted string.

089 Syntax error in PSECT declaration

Explanation: The psect declaration is improperly coded.

User Action: Check and correct the declaration.

090 Missing semicolon or TES following a CASE action

Explanation: Each case action expression in a list must be followed by a semicolon and the list must be concluded by TES.

User Action: Insert the missing semicolons or the keyword TES.

091 No CASE-label matches the CASE-index

Explanation: Based on its evaluations of the low to high bound and case label values, the compiler has determined that for the values of the case index no selector element will be matched.

User Action: Evaluate the case index and its bounds relative to the values of the case labels, or include INRANGE and OUTRANGE labels.

092 Some values in the range given by FROM and TO have no matching CASE-label

Explanation: The compiler cannot match all of the low to high bound values with the case label values given.

User Action: Evaluate the case label values relative to those of the low to high bound values.

093 No structure attribute for variable <name> in structure reference

Explanation: The variable name shown has been declared, but the structure attribute is missing.

User Action: Insert the appropriate structure attribute (structure name and allocation actuals) for the declaration of the data segment named.

094 Routine specified as MAIN is not defined

Explanation: The routine name specified in the MAIN switch also must be defined by a routine or global routine declaration in the same module.

User Action: Define the routine with the appropriate declaration.

095 %REF built-in function must be used only as a routine actual parameter

Explanation: Built-in function has been used improperly.

User Action: Correct the use of the function.

096 Module body contains executable expression or non-link-time constant declaration

Explanation: An executable expression (such as .x) or a non-link-time constant expression declaration should not appear within the outermost level of the module body.

User Action: Correct the use of expressions and declarations within the outermost level of the module body.

097 Length of quoted string parameter of <lexical-function> must not exceed <integer-value>

Explanation: The quoted string of the function shown must not contain more characters than the value shown.

User Action: Correct the length of the parameter.

098 Cannot satisfy REGISTER declarations

Explanation: Too many registers (in linkage, globals, built-in or predeclared functions) are simultaneously active.

User Action: Redistribute explicit register usage to prevent overlaps in time.

099 Simultaneously allocated two quantities to Register <integer-value>

Explanation: Two conflicting data segments have been allocated at the same time for the register shown.

User Action: Correct the data segment allocations.

100 Division by zero

Explanation: An illegal arithmetic operation has been performed.

User Action: Correct the operation.

101 Name to be declared is missing

Explanation: A name has not been specified in the declaration.

User Action: Specify a name in the declaration.

102 Null structure actual parameter <name> has no default value

Explanation: A null reference has been made with an access actual expression for which no default value exists.

User Action: Specify a value in the access actual expression.

103 Illegal up-level reference to <name>

Explanation: Reference has been illegally made from a nested routine declaration to a name in a higher level block. References are not permitted to LOCAL, REGISTER, or STACKLOCAL storage that is declared in a routine declaration which contains the routine declaration currently being compiled.

User Action: Delete and relocate the reference or the name to an appropriate block.

104 Missing ELUDOM following module

Explanation: The end module keyword is missing.

User Action: Insert the ELUDOM keyword at the end of the module.

105 Language feature not yet implemented in <language>: <feature-keyword-name>

Explanation: Language feature shown is not yet supported in this dialect.

User Action: Remove the language feature from the program.

106 REQUIRE file nesting depth exceeds implementation limit of 9

Explanation: Require declarations or lexical functions have been nested beyond allowable limit.

User Action: Reconfigure nesting within allowable limits.

107 Structure and allocation-unit or extension are mutually exclusive

Explanation: An allocation unit attribute or an extension attribute cannot appear with a structure attribute in an allocation declaration.

User Action: Remove the contradictory attributes.

108 Allocation-unit must not follow INITIAL attribute

Explanation: The allocation unit attribute must precede the initial attribute in a declaration.

User Action: Rearrange the order of the attributes.

109 Missing quoted string following REQUIRE or LIBRARY

Explanation: Quoted file name not found in a require or library declaration.

User Action: Insert and/or quote file name in declaration.

110 Open failure for REQUIRE or LIBRARY file

Explanation: The file specified in a require or library declaration cannot be accessed by the compiler.

User Action: Check validity of file name or make file available to compiler.

111 Comment not terminated before end of <source-file-name>

Explanation: An embedded comment must end with a close parenthesis and a percent sign, and the comment must end in the same source file in which it began.

User Action: Correct the insertion of the imbedded comment.

112 Definition of macro <macro-name> not terminated before end of program

Explanation: A macro declaration must be terminated by a percent sign followed by a semicolon.

User Action: Terminate the macro name shown.

113 Missing semicolon, right parenthesis or END following a subexpression of a block

Explanation: Each subexpression must be concluded by a semi-colon and the block must be concluded with a close parenthesis or an END.

User Action: Insert the appropriate terminators.

114 Invalid REQUIRE or LIBRARY file specification

Explanation: The specified require file must be a valid name to the compiler and the system, and the library file must be a binary file produced by the correct compiler dialect.

User Action: Check and correct the validity of the file.

115 Expression identified by a label must be a block

Explanation: A labeled expression must be contained within a BEGIN-END or parenthesis pair.

User Action: Enclose the expressions within a block.

116 Value of structure size-expression must be a compile-time constant

Explanation: The size expression must meet the criteria for a compile-time constant expression.

User Action: Evaluate and correct the size expression.

117 Value of structure size-expression must not be negative

Explanation: A structure size-expression must not indicate a negative value.

User Action: Evaluate and correct the size expression value.

118 Missing left parenthesis in PLIT or INITIAL attribute

Explanation: A PLIT item or an initial attribute must be enclosed in parentheses.

User Action: Insert the missing open parenthesis.

119 ALWAYS illegal in a SELECTONE expression

Explanation: The select label ALWAYS cannot be used with a SELECTONE, SELECTONU, or SELECTONEA expression.

User Action: Correct the select expression.

120 Case range spanned by FROM and TO exceeds implementation limit of 512

Explanation: Range of case expression cannot exceed the high bound limit of 512.

User Action: Evaluate and correct the range values.

121 Percent sign outside macro declaration

Explanation: An improperly quoted (%QUOTE) percent sign is contained in a nested macro declaration, or an extra percent sign has been found in the source file.

User Action: Evaluate and correct the use of the percent sign for the macro declaration.

122 Recursive invocation of non-recursive macro <macro-name>

Explanation: Only a conditional macro with one or more formal names can be used recursively.

User Action: Correct the definition of the macro named.

123 Recursive invocation of structure <structure-name>

Explanation: A structure cannot invoke itself directly or indirectly.

User Action: Correct the declaration of the structure named.

124 Expression nesting or size of a block exceeds implementation limit of 300

Explanation: More expressions have been nested or a block contains more lines than are allowed.

User Action: Decrease the number of nested expressions or the number of lines in the block.

125 Operand preceding left bracket in structure reference is not a variable name

Explanation: The operand preceding the access actual parameter must be a variable name.

User Action: Evaluate and correct the operand.

126 Value of PLIT replicator must not be negative

Explanation: The REP replicator must be a compile-time constant expression that does not indicate a negative value.

User Action: Evaluate and correct the replicator value.

127 RETURN not within a routine

Explanation: To properly return control to the caller, the return expression must be enclosed within the BEGIN-END pair of the called routine.

User Action: Correct the placement of the return expression within the outermost level of the routine, or check for the exclusion of the END keyword from the routine.

128 BIND or LITERAL name <name> used in its own definition

Explanation: The data name value for a bind declaration or the literal value for a literal declaration must not contain a name already declared bind or literal.

User Action: Evaluate name shown and correct its declaration.

129 Missing comma or right parenthesis in actual parameter list for <routine-or-macro-name>

Explanation: Each actual parameter in a call list must be separated by a comma and the list must be ended by a close parenthesis.

User Action: Insert the missing commas or close parenthesis in the call to the routine or macro named.

130 Omitted actual parameter in call to <keyword-macro-name> has no default value

Explanation: In reference call to keyword macro named, no default value exists for the omitted actual parameter.

User Action: Provide an appropriate value for the omitted actual parameter.

131 Extra actual parameters in call to <builtin-function-name>

Explanation: The number of actual parameters used in call to a built-in function must not exceed the number of formal parameters used in the built-in routine.

User Action: Correct actual parameter usage in call to built-in function named.

132 Translation table entries in call to CH$TRANSTABLE must be compile-time constants

Explanation: The translation items do not meet the criteria for compile-time constant expressions.

User Action: Evaluate and correct the translation items in the call.

133 Allocation unit (other than BYTE) in call to CH$TRANSTABLE

Explanation: Character positions in a translation table are restricted to the length of a byte.

User Action: If an allocation unit attribute is necessary, insert the keyword BYTE.

134 Length of table produced by CH$TRANSTABLE (<integer-value>) not an even number between 0 and 256

Explanation: The number of translation items used in the call must be even.

User Action: Reduce or increase the length of the table by an even number that is closest to the number of character positions desired.

135 Length of destination shorter than sum of source lengths in CH$COPY

Explanation: The sum of the source length parameters (sn1+sn2+...) must not be greater than the value of the destination parameter (dn).

User Action: Increase the value of the destination parameter.

136 Character-size parameter of <character-function-name> must be equal to 8

Explanation: The character function named has illegally specified a character size other than eight bits in length; only BLISS–36 supports character sizes other then eight.

User Action: Insert a character size value of eight.

137 Built-in routine has no value

Explanation: A machine specific function that cannot produce a value has been used in a context where a value is required.

User Action: Evaluate the required use of the built-in function and correct the coding.

138 Missing equal sign in GLOBAL REGISTER declaration

Explanation: The global register declaration is incomplete.

User Action: Insert the missing equal sign following the register name.

139 Illegal use of %REF built-in function as actual parameter <integer-value> of call to <routine-name>

Explanation: The value of a %REF function is the address of a temporary data segment which stores a copy of the value of the actual parameter; thus its use is often incompatible with the storage requirements of a built-in function.

User Action: Delete %REF and provide a call to the routine named that will provide permanent storage for the value returned.

140 Illegal use of register name as actual parameter <number> of call to routine <routine-name>

Explanation: An undotted register name has been used as an actual parameter for the routine call shown

User Action: Provide a legal register name.

141 Routine <routine-name> has no value

Explanation: The mechanism for returning a value is suppressed.

User Action: Remove the novalue attribute from the routine named.

142 Missing quoted string following CODECOMMENT

Explanation: A quoted string is required for each comment.

User Action: Enclose the affected comments in quotes.

143 Missing comma or colon following CODECOMMENT

Explanation: Each quoted string in the list must be separated by a comma and the list must be ended with a colon.

User Action: Insert the missing commas and/or the colon.

144 Expression following CODECOMMENT must be a block

Explanation: The expression following the colon must be enclosed with a parenthesis or BEGIN-END pair.

User Action: Enclose the expression appropriately.

145 Illegal OPTLEVEL value <value>

Explanation: The only valid optimization level values are zero through 3.

User Action: Replace the switch value shown with an appropriate value.

146 ENABLE declaration must be in outermost block of a routine

Explanation: The enable declaration must reside in the outermost level of the establisher routine.

User Action: Correct the placement of the enable declaration.

147 More than one ENABLE declaration in a routine

Explanation: An establisher routine must not enable more than one handler routine.

User Action: Remove all but one of the enable declarations.

148 Handler specified by ENABLE must be a routine name

Explanation: The name specified by an enable declaration must be the name of a routine.

User Action: Provide an appropriate routine name for the declaration.

149 Illegal actual parameter in ENABLE declaration

Explanation: Actual parameters for enable declarations are restricted in use to names declared as own, global, forward, or local names.

User Action: Provide an appropriately declared name for the actual parameter.

150 Name used as ENABLE actual parameter must be VOLATILE: <name>

Explanation: A volatile attribute must be used to warn the compiler that the declared actual parameter is subject to unexpected change.

User Action: Provide a volatile attribute for the actual parameter named.

151 Missing comma or right parenthesis in ENABLE actual parameter list

Explanation: Each actual parameter in a list must be separated by a comma and the list must be ended with a close parenthesis.

User Action: Insert the missing commas and/or close parenthesis.

152 LANGUAGE switch specification excludes <language-name>

Explanation: The language name shown is missing from the language list in a switch declaration.

User Action: Insert the missing language name.

153 Missing OF following REP

Explanation: The replicator construct for the expression is incomplete.

User Action: Insert the missing keyword OF following the replicator.

154 Incorrect number of parameters in call to lexical function <lexical-function-name>

Explanation: A lexical function must conform to its syntactic definition.

User Action: Evaluate and correct parameter usage for lexical function named.

155 Number of parameters of ENTRY switch exceeds implementation limit of 128

Explanation: The module switch has too many parameters.

User Action: Reduce the number of parameters used in the ENTRY switch.

156 Unknown name in BUILTIN declaration: <name>

Explanation: Only a name predefined for BLISS can be declared as built-in.

User Action: Correct the name shown or delete it or use another form of declaration for it.

157 Conditional out of sequence: <name>

Explanation: The keyword named is improperly sequenced in the lexical conditional.

User Action: Evaluate and correct the order in which the keywords appear in the expression.

158 <%PRINT, %INFORM, %WARN, %ERROR, or %ERRORMACRO>: <advisory-text>

Explanation: This is the form of the message number and text that appears when one of the lexical functions shown is used.

User Action: Example: INFO 158,%INFORM: 'user text specified by function'

159 Conditional not terminated before end of <macro or source-file-name>

Explanation: Lexical conditional is not properly terminated in the file named.

User Action: Insert the missing termination keyword %FI.

160 Missing formal parameter or equal sign in call to keyword macro <macro-name>

Explanation: Each macro actual parameter in a keyword macro call must be connected by an equal sign to a keyword formal name previously declared in a keyword macro.

User Action: Insert the missing formal name or the missing equal sign.

161 Formal parameter <parameter-name> multiply specified in call to keyword macro <macro-name>

Explanation: In a keyword macro call to the macro named, the multiplication of the keyword formal name shown has been illegally specified.

User Action: Evaluate and correct the call.

162 Missing %THEN following %IF

Explanation: The coding of a lexical conditional is incomplete.

User Action: Insert the missing required keyword %THEN.

163 Actual parameter <parameter-name> of call to routine <routine-name> is illegal

Explanation: An invalid actual parameter has been used in a call to the routine named.

User Action: Evaluate and correct the use of actual parameter named in the call.

164 Language feature to be removed: <feature>

Explanation: Compiler reports that the use of feature named is discontinued.

User Action: Evaluate and correct module.

165 Language feature not present in <language>: <feature>

Explanation: The compiler reports that the feature named is not available to the dialect named.

User Action: Remove the feature from the module switch for the dialect named.

166 Name declared STACK is not properly defined

Explanation: The name used as a stack data segment has not been declared.

User Action: Correct or define name with stacklocal declaration.

167 Name declared ENTRY is not globally defined: <name>

Explanation: In BLISS–36, name shown has been designated for entry in global object module record and has not been declared as global.

User Action: Define name shown with global declaration.

169 Fetch or store applied to field of zero size

Explanation: Attempted fetch or assignment expression to an invalid data segment.

User Action: Correct range attribute for data or structure declaration.

170 Missing equal sign in FIELD declaration

Explanation: An equal sign must appear between the field set name and the keyword SET and between each field name and the left bracket of the field component.

User Action: Insert missing equal signs.

171 Missing comma on right bracket in FIELD declaration

Explanation: Comma must appear after right bracket of each field definition (except the last) in list.

User Action: Insert missing commas.

172 Missing left bracket in FIELD declaration

Explanation: A left bracket must appear before each list of field components in a list of field definitions.

User Action: Insert missing left brackets.

173 Missing comma or TES in FIELD declaration

Explanation: A comma must appear between each field component in a list and the list must be ended with TES.

User Action: Insert missing commas or keyword TES.

174 Missing left bracket or SET in FIELD declaration

Explanation: The equal sign following the field set name must be followed by a SET, and each equal sign following a field name must be followed by a left bracket.

User Action: Insert keyword SET or left brackets.

175 Number of field components exceeds implementation limit of 32

Explanation: The number of components in a field definition exceeds the limits allowed for a structure.

User Action: Decrease the number of components or create separate structures.

176 Field name <name> invalid in structure reference to variable <variable-name>

Explanation: The field name shown as an access actual parameter does not agree with the variable name shown as a field declaration.

User Action: Evaluate and correct the uses of name.

177 Parameter of FIELD attribute must be a field or field-set name

Explanation: Invalid parameter has been used for a field attribute; the name used must be identified by a field declaration as a field or field set name.

User Action: Replace parameter with a declared field or field set name.

178 Number of parameters of FIELD attribute exceeds implementation limit of 128

Explanation: Excessive number of field names have been specified in field attribute.

User Action: Decrease the number of field names or declare a field set for the number in excess.

179 Missing equal sign in LINKAGE declaration

Explanation: An equal sign must appear between a linkage name and a linkage type.

User Action: Insert the missing equal sign.

180 Invalid linkage type specified

Explanation: The linkage type specified for the dialect is illegal.

User Action: Evaluate and correct the linkage type word.

181 Illegal register number <integer> in LINKAGE declaration

Explanation: The register number shown is invalid.

User Action: Evaluate and correct the register number.

182 Multiple specification of register <register-number> in LINKAGE declaration

Explanation: The register shown has been specified more than once in the declaration.

User Action: Evaluate and correct register specifications.

183 Invalid parameter location specified

Explanation: The parameter location specified is illegal.

User Action: Check the legal uses of parameter locations and correct the specifications.

184 Missing comma or right parenthesis in LINKAGE declaration

Explanation: Each parameter location in a list must be separated by a comma and the list must be ended by a close parenthesis.

User Action: Insert the missing commas and/or the close parenthesis.

185 Invalid linkage modifier in LINKAGE declaration

Explanation: An illegal modifier has been used as a linkage option.

User Action: Check and correct the use of linkage option modifiers for the dialect.

186 Missing left parenthesis in LINKAGE declaration

Explanation: A parameter location list must be preceded by an open parenthesis.

User Action: Insert the missing open parenthesis.

187 Missing global register name in LINKAGE declaration

Explanation: A global linkage option has been used and the global register name has not been specified.

User Action: Insert the missing global register name.

188 No match in linkage <name> for EXTERNAL REGISTER variable <name>

Explanation: The register named in the global linkage option must be the same as the register named in the associated external register declaration.

User Action: Use the same register name in both the routine and its linkage declaration.

189 Global register <name> specified by linkage <linkage-name> not declared to call

Explanation: The register named in the global linkage option has not been declared in a call to the routine.

User Action: Declare the register name within the calling routine with an external register declaration.

190 WORD or Radix–50 item number <integer> allocated at odd byte boundary

Explanation: Data structure is improperly allocated.

User Action: Correct data allocation to place WORD or RAD50_11 value shown at a word boundary.

191 Multiple GLOBAL declaration of name: <name>

Explanation: The global name shown has been declared more than once in the same module.

User Action: Delete all the extra appearances of the global declaration.

192 Multiple declaration of name in assembly source: <name>

Explanation: The name shown has been declared more than once in a module that was compiled with the assembleable listing option.

User Action: If the intent is to run the listing through an assembler, delete all extra appearances of the declared name; or, use the switch item UNAMES in a switches declaration to obtain unique names.

193 <declaration-name> declaration not available when OBJECT(ABSOLUTE) in effect

Explanation: This message is reserved for BLISS–16 future expansion.

User Action: No action is required.

194 Library source module must contain only declarations

Explanation: Executable expressions must not appear in a library source file.

User Action: Remove all but the declarations from the library source file.

195 LIBRARY file has invalid format

Explanation: The internal formatting of the file is incorrect.

User Action: The specified file is probably not a precompiled library file; change the file specification and recompile. If the problem persists, submit an SPR.

196 LIBRARY file must be regenerated using current compiler release

Explanation: A library source file must be precompiled again using the latest version of the compiler.

User Action: Use the latest version of the compiler to regenerate the library file.

197 LIBRARY file must be generated using <language>

Explanation: The library file must be precompiled by the compiler associated with the dialect named.

User Action: Generate the library file with the compiler associated with the dialect named.

198 LIBRARY file contains internal consistency error

Explanation: A library file has been referenced that has been precompiled with errors.

User Action: Recompile the library source file with a /LIBRARY qualifier, and if the problem persists, submit an SPR.

199 Warnings issued during LIBRARY precompilation: <number>

Explanation: The number shown is the number of warnings issued during the precompilation of the file.

User Action: Evaluate and correct all warnings and recompile.

200 Illegal declaration type in library source module

Explanation: Only certain types of declarations can be used in a library source file.

User Action: Remove the invalid declarations from the library source file and regenerate the file.

201 Illegal occurrence of bound name <name> in library source module

Explanation: Bound names cannot be inserted in library source file.

User Action: Remove the declaration for the name shown from the file and regenerate the file.

202 Number of parameters of ARGTYPE linkage attribute modifier exceeds implementation limit of 128

Explanation: Excessive number of parameters used with built-in linkage function ARGTYPE.

User Action: Reduce the parameters to an acceptable number.

203 <name> linkage modifier not available with this linkage type

Explanation: The linkage option named cannot be used with the linkage type specified.

User Action: Evaluate and use an appropriate linkage option.

204 Length of SYSLOCAL specification not in range 1 to 15

Explanation: This message reflects a future enhancement.

User Action: No action is required.

205 BUILTIN declaration of <name> invalid in this context

Explanation: Each name used in a built-in declaration must be predefined (but not predeclared); however, if a register name or linkage function is used, it must also be contained in a routine declaration.

User Action: Evaluate and correct the use of the name shown.

206 BUILTIN operation needs a register declared as NOTUSED

Explanation: A register required by a built-in function is unavailable for use due to a NOTUSED linkage modifier.

User Action: Delete or change the modifier in the associated linkage declaration to allow the register to be used.

207 NOTUSED linkage modifier of caller is not a subset of that of called routine

Explanation: The linkage type and linkage option of the caller routine is incompatible with that of the called routine.

User Action: Evaluate and correct the linkage declarations.

208 Called routine does not preserve register declared NOTUSED by caller

Explanation: To preserve all the necessary registers, all of the locally usable registers of the called routine must be declared as locally usable registers in the caller routine.

User Action: Evaluate and correct the linkage declarations.

209 Illegal character or field too large in VERSION

Explanation: The quoted string in the VERSION switch must conform to the TOPS–10/20 version number format oooa(oooooo)-o; where "o" is an octal digit and "a" is an alphabetic character.

User Action: Correct the string in regard to the version number format.

210 Stack pointers in different registers

Explanation: The number of the stack pointer register is assigned by default and depends on the dialect used; however, the default number of the register can be altered by a change in the declared linkage type (such as F10) while neglecting to specify the LINKAGE_REGS option.

User Action: Specify the desired stack pointer register number by using a LINKAGE_REGS modifier for the altered linkage type.

211 Use of uninitialized data-segment <name>

Explanation: An attempt has been made to use the data segment named without first initializing it.

User Action: Insert an initial attribute in the data segment declaration, or assign a value to the segment before fetching from it.

212 Null expression appears in value-required context

Explanation: A null expression has been used where a value is required.

User Action: Evaluate and provide a value for the expression.

213 Expressions eliminated following RETURN, LEAVE or EXITLOOP

Explanation: These expressions end the evaluation of a routine body (return), a block (leave), or an innermost loop (exitloop); therefore, they must be the last expressions inserted before the affected block is ended, if not all subsequent expressions will not be compiled.

User Action: Evaluate and correct the insertion of the return or exit expression.

214 Language feature not transportable

Explanation: The feature specified for the dialects defined by the language switch is not transportable.

User Action: Evaluate the feature and take appropriate action.

215 Language feature not transportable: <name>

Explanation: The feature specified by the name shown is not transportable to the dialects defined by the language switch.

User Action: Evaluate the feature named and take appropriate action.

216 Language feature not transportable: <keyword>

Explanation: The feature specified by the keyword shown is not transportable to the dialects defined by the language switch.

User Action: Evaluate the feature shown and take appropriate action.

217 GLOBAL or EXTERNAL name not unique in 6 characters: <name>

Explanation: In BLISS–16 and BLISS–36, at least six characters in a global or external name must be unique.

User Action: Evaluate and correct the global or external name.

218 Implicit declaration of BUILTIN <linkage-name> to be withdrawn

Explanation: The compiler implicitly declares the function named as built-in when a FORTRAN linkage routine is being compiled.

User Action: Add an explicit built-in declaration within the proper scope.

219 Empty compound expression is illegal

Explanation: A compound expression block does not contain any declarations, but it must contain at least one expression to be legal.

User Action: Insert an expression in the block or delete the entire block form.

220 PRESET items have overlapping initialization

Explanation: A preset value must not occupy more storage than is allocated for the data segment, and the field names described in the preset items must not overlap.

User Action: Evaluate and correct the preset attribute.

221 Missing left square-bracket in PRESET attribute

Explanation: Each preset item in a preset attribute must be preceded by an open bracket.

User Action: Insert the missing open bracket before the preset item.

222 Source line too long. Truncated to 132 characters.

Explanation: A line in the source file exceeds the implementation limit of 132 characters.

User Action: Decrease the size of the source line.

223 Name used in routine-call not declared as ROUTINE: <routine-name>

Explanation: The routine designator used in a routine call must yield a value declared as a routine name in a routine declaration.

User Action: Ensure that the name used in the routine call is declared in a routine declaration.

224 INTERRUPT general routine call is invalid

Explanation: A linkage name defined by an INTERRUPT linkage type must not be used with this dialect in a general routine call.

User Action: With this dialect, use an ordinary routine call to invoke the interrupt routine.

225 Invalid linkage attribute specified <attribute-name> is assumed

Explanation: A linkage attribute must be either a predeclared linkage name or one specified in a linkage declaration.

User Action: Evaluate and correct the use of the linkage name in the linkage attribute.

226 Value of a linkage name <name> is outside permitted range

Explanation: The value of the linkage name shown exceeds the compatible and transportable range of the dialect.

User Action: Provide a linkage name that is within the compatible and transportable range of the dialect.

227 Effective position and size outside of permitted range

Explanation: The values of the field reference parameters have exceeded the structure allocation specified for the data segment.

User Action: Evaluate and correct the value of the offset and field size parameters.

228 Builtin machop <name> has no value

Explanation: The instruction named did not produce a value when executed by the machine specific function MACHOP.

User Action: Select a machine instruction that will produce a value when executed.

229 Parameter <parameter-name> of builtin <name> has value outside the range

Explanation: The parameter named for the built-in function named indicates a value that exceeds the specified range.

User Action: Decrease the value of the parameter named to conform with the specifications of the function named.

230 Parameter <parameter-name> of builtin <name> must be a link-time constant expression

Explanation: The parameter named for the built-in function named is an invalid expression.

User Action: Replace the parameter named with an expression that meets the criteria for a link time constant.

231 Invalid linkage attribute specified CLEARSTACK is added

Explanation: The CLEARSTACK linkage option is illegal with this dialect.

User Action: Delete the CLEARSTACK modifier from the linkage declaration.

232 OTS linkage specified twice

Explanation: The OTS option of the ENVIRONMENT switch specifies the use of a standard OTS file and linkage; therefore, the switch must not appear in the same module with an OTS switch and an OTSLINKAGE switch which specifies the use of a nonstandard file and linkage.

User Action: Evaluate and correct the coding for OTS.

233 OTS linkage <name> not declared before first routine declaration

Explanation: The linkage name specified by the OTSLINKAGE switch must be predeclared or appear in a linkage declaration that precedes the first routine declaration in the module.

User Action: Define the linkage name shown in a linkage declaration that precedes the first routine declaration in the module.

234 OTS linkage <name> may not use global registers or pass parameters by register

Explanation: The linkage name specified by the OTSLINKAGE switch must not specify register or global register parameter locations.

User Action: Evaluate and correct the use of the parameter locations in the linkage declaration named.

235 OTS linkage <name> not defined before it's used

Explanation: The linkage name shown has not been declared prior to its use in an OTSLINKAGE switch.

User Action: Declare the linkage name shown with a linkage declaration.

236 First PSECT declaration appears after a declaration that allocates storage

Explanation: In BLISS–36, the first psect declaration in a module must appear before the first declaration that causes storage to be allocated or object code to be generated.

User Action: Reinsert the first psect declaration before the first data or routine declaration (external and forward types excepted) and/or the first PLIT expression in the module.

237 Exponent for floating or double floating literal out of range

Explanation: Exponent value is too large for floating literal.

User Action: Evaluate and correct the value of the exponent.

239 String exceeding implementation limits (<number> characters) was truncated

Explanation: The string function (such as %EXACTSTRING) exceeds the implementation limit of 1000 characters for the length of a sequence.

User Action: Decrease the size of the string.

240 <reserved-word> declaration is illegal in STRUCTURE declaration

Explanation: The declaration defined by the reserved word shown (such as OWN) is illegal in a structure declaration.

User Action: Remove the illegal declaration.

242 Output formal parameter <name> in routine declaration was not described in linkage

Explanation: An output parameter location has not been specified in the corresponding linkage declaration for the output formal parameter shown.

User Action: Specify the output parameter location for the output formal parameter named in the routine declaration.

243 Output actual parameter was not described in linkage

Explanation: An output parameter location has not been specified in the corresponding linkage declaration for an output actual parameter specified in the caller routine.

User Action: Specify an output parameter location for the output actual parameter specified in the caller routine.

244 Name declared UNDECLARE is not defined:<name>

Explanation: An undeclare declaration has been used with a name that has not been declared.

User Action: Declare the name shown.

246 FORWARD declaration of <name> cannot be satisfied by BIND declaration

Explanation: A name declared as FORWARD must be defined as a ROUTINE, OWN, or GLOBAL name.

User Action: Evaluate and correct the use of the specified name on the FORWARD declaration.

247 Character size parameter of <name> must be equal to a compile-time constant in the range 1 to 36

Explanation: The character size value of the named character function is either outside of the permissible range or not a compile-time constant.

User Action: Provide a compile-time constant within the permissible range.

248 <number> is an illegal character size for a global byte pointer. A local byte pointer will be generated

Explanation: A program with extended addressing was compiled having a CH$PTR function size value that is invalid for creating a global byte pointer.

User Action: Determine if a local byte pointer is acceptable; if not, change the size value to reflect a valid global byte pointer.

249 EXTENDED addressing is not supported under TOPS–10

Explanation: The compiler has reported that it cannot support extended addressing under TOPS–10.

User Action: Remove the extended addressing feature from the program.

250 Referenced <LOCAL/STACKLOCAL/REGISTER> symbol <name> is probably not initialized.

Explanation: This indicates a reference to the value of a local or register symbol (with a dot) before that symbol has received a value. This message may be printed erroneously when the initialization of a variable occurs on one branch of a conditional (IF, CASE, or SELECT), and a reference to the variable occurs on another branch of the conditional, and the conditional is contained in a loop.

Note that this error message may indicate the occurrence of a common programmer mistake whereby the address of a variable is stored in a control block and the control block is then passed to a routine (such as a system service or RMS routine) which will update the data at that address. If the variable does not have the VOLATILE attribute, the compiler cannot know that its value is potentially changed by the routine call.

User Action: Initialize the symbol prior to its first use.

251 Symbol <name> is declared <class> in an outer block

Explanation: A symbol name declared in an outer block is inaccessible because it is also declared in an inner block.

User Action: Ensure that all references to the name in the inner block refer to the symbol declared there and not to the one declared in the outer block.

252 Test expression is always <true/false>

Explanation: During the optimization of an IF, WHILE, or UNTIL structure, a test expression has been reduced to a constant with the possible elimination of code.

User Action: Evaluate the test expression for proper operation.

253 Action <number> <never/always> true. <elimination-text>

Explanation: One or more of the action statements in a SELECT or SELECTONE construct cannot be reached, and consequently certain actions have been eliminated.

User Action: Evaluate the select statements for proper operation.

254 PRESET-attribute must be preceded by a structure-attribute

Explanation: To use a PRESET attribute to initialize the fields of a structured data segment, there must be a structure attribute to associate the name of the data segment specified in the preset item with a structure definition.

User Action: Correct the use of the PRESET attribute.

255 Offset in PRESET-item cannot be negative

Explanation: The PRESET item should be a positive value.

User Action: Correct the value of the item.

256 Filespec TRUNCATED, too long for DEBUG DST/SFC Record: <filespec>

Explanation: The filespec is too long to fit in the record that describes it to the debugger.

User Action: Rename the file using a shorter name.

257 /MASTER_CROSS_REFERENCE qualifier has been superseded by /ANALYSIS_DATA

Explanation: The /MASTER_CROSS_REFERENCE qualifier is obsolete. The /ANALYSIS_DATA qualifier now allows you to extract an expanded version of such information and use the VAX Source Code Analyzer (SCA) to interactively query or report it.

User Action: Use the /ANALYSIS_DATA qualifier to obtain comprehensive cross-reference data.

259 Number of psects used exceeds implementation limit of 32768

Explanation: The number of psects used in the current module exceeds the number allowed.

User Action: Reduce the number of psects to the required limit.

E.1 BLISS Compiler Fatal Errors

The following fatal error messages indicate serious problems with the environment or the compiler, or both. When such a condition is detected, compilation terminates immediately.

INTERNAL COMPILER ERROR

The compiler has failed an internal consistency check. This message may be followed by an error number. BLISS–32 then issues a traceback printout. (BLISS–36 is unable to issue a traceback.) Please submit an SPR and include a copy of the program that generated this message.

INSUFFICIENT DYNAMIC MEMORY AVAILABLE

On the VAX, this error may indicate a bug in the compiler. On the DECsystem-10 /20, your program may be too large to compile.

I/O ERROR ON INPUT FILE
An error occurred while an input file was being accessed. This may be preceded by other error messages that provide more specific information about the error.

I/O ERROR ON OBJECT FILE
An error occurred while the output object file was being accessed. This may be preceded by other error messages that provide more specific information about the error.

I/O ERROR ON LISTING FILE
An error occurred while the output listing file was being accessed. This may be preceded by other error messages that provide more specific information about the error.

I/O ERROR ON LIBRARY FILE
An error occurred while a BLISS precompiled library file was being accessed. This may be preceded by other error messages that provide more specific information about the error.

LIBRARY PRE-COMPILATION EXCEEDS COMPILER LIMIT
A precompiled BLISS library cannot be larger than approximately 2048 (DECsystem-10/20) disk blocks; the library file will be deleted. VAX libraries have no restrictions.

MACRO OR STRUCTURE DECLARATION WITHIN STRUCTURE BODY
This is a permanent implementation restriction in the BLISS language.

REQUIRE DECLARATION WITHIN MACRO BODY
This is a permanent implementation restriction in the BLISS language.

FATAL ERROR IN COMMAND LINE
This message appears on VMS only, for BLISS–32 or BLISS–16. Your command line was improperly formed. A previous error message provided additional information to describe what was wrong.

I/O ERROR DURING COMMAND LINE SCANNING
This message appears on TOPS–10 or TOPS–20 when a severe error is encountered in command line parsing.

NESTED EXPRESSION TOO DEEP. SIMPLIFY AND RECOMPILE
The source program contains more than 64 levels of nested blocks, each containing declarations.

UNRECOVERABLE SOURCE ERRORS. CORRECT AND RECOMPILE
This message appears on VMS only, for BLISS–32 or BLISS–16. Errors previously encountered by the compiler have confused it to the point at which it cannot continue the compilation.

F

Sample Output Listing

Example F–1 contains the complete output listing for the module TESTFACT. Chapter 2 examples use excerpts from this listing.

Example F–1 Sample Output Listing

```
TESTFACT    11-Mar-1987 15:58:48    VAX BLISS--32 V4.3                  Page   1
            22-Jul-1983 16:06:56    DISK$:[DIRECTORY]MYPROG.B32;13          (1)

;   0001  0     MODULE TESTFACT (MAIN = MAINPROG)
;   0002  0     BEGIN
; WARN#048      1  L1:0002
; Syntax error in module head
;   0003  1
;   0004  1     OWN
;   0005  1         A,
;   0006  1         B;
;   0007  1
;   0008  1     ROUTINE IFACT (N) =
;   0009  2         BEGIN
;   0010  2         LOCAL
;   0011  2             RESULT;
;   0012  2         RESULT = 1;
;   0013  2         INCR I FROM 2 TO .N DO
;   0014  2             RESULT = .REULT*.I;
; WARN#000      ..................1  L1:0014
; Undeclared name:  REULT
;   0015  2         .RESULT
;   0016  1         END;

                                              .TITLE  TESTFACT

                                              .PSECT  $OWN$,NOEXE,2

                                00000 A:      .BLKB   4
                                00004 B:      .BLKB   4

                                              .EXTRN  REULT

                                              .PSECT  $CODE$,NOWRT,2

                           0000 00000 IFACT:  .WORD   Save nothing        ; 0008
              50         01  D0 00002         MOVL    #1, RESULT          ; 0012
              51         01  D0 00005         MOVL    #1, I               ; 0013
                         06  11 00008         BRB     2$                  ;
50     0000G  CF         51  C5 0000A 1$:     MULL3   I, REULT, RESULT    ; 0014
F5            51    04   AC  F3 00010 2$:     AOBLEQ  N, I, 1$            ;
                             04 00015         RET                         ; 0016

; Routine Size:  22 bytes,    Routine Base:  $CODE$ + 0000
```

(continued on next page)

Example F–1 (Cont.) Sample Output Listing

```
;   0017  1
;   0018  1     ROUTINE RFACT (N) =
;   0019  1         IF .N GTR 1
;   0020  1        THEN
;   0021  1             .N * RFACT(.N - 1)
;   0022  1         ELSE
;   0023  1             1;


                          0000 00000 RFACT:  .WORD   Save nothing       ; 0018


TESTFACT    11-Mar-1987 15:58:48    VAX BLISS--32 V4.3                 Page   2
            22-Jul-1983 16:06:56    DISK$:[DIRECTORY]MYPROG.B32;13         (1)

             01       04   AC  D1 00002         CMPL    N, #1           ; 0019
                           0E  15 00006         BLEQ    1$              ;
7E       04   AC           01  C3 00008         SUBL3   #1, N, -(SP)    ; 0021
         EF   AF           01  FB 0000D         CALLS   #1, RFACT       ;
             50       04   AC  C4 00011         MULL2   N, R0           ;
                               04 00015         RET                     ;
             50            01  D0 00016 1$:     MOVL    #1, R0          ; 0019
                               04 00019         RET                     ; 0023

; Routine Size:  26 bytes,    Routine Base:  $CODE$ + 0016

;   0024  1
;   0025  1     ROUTINE MAINPROG =
;   0026  2         BEGIN
;   0027  2         A = IFACT(5);
;   0028  2         B = RFACT(5);
;   0029  2
;   0030  2         1                         ! VMS wants a success return
;   0031  1         END;



                       0000 00000 MAINPROG:
                                           .WORD   Save nothing         ; 0025
                     05  DD 00002          PUSHL   #5                   ; 0027
   C8   AF           01  FB 00004          CALLS   #1, IFACT            ;
 0000'  CF           50  D0 00008          MOVL    R0, A                ;
                     05  DD 0000D          PUSHL   #5                   ; 0028
   D3   AF           01  FB 0000F          CALLS   #1, RFACT            ;
 0000'  CF           50  D0 00013          MOVL    R0, B                ;
        50           01  D0 00018          MOVL    #1, R0               ; 0031
                         04 0001B          RET                          ;

; Routine Size:  28 bytes,    Routine Base:  $CODE$ + 0030

;   0032  1     END
;   0033  0     ELUDOM
```

(continued on next page)

Example F–1 (Cont.) Sample Output Listing

```
;                                      PSECT SUMMARY
;
;  Name       Bytes                         Attributes
;
;  $OWN$       8  NOVEC,  WRT,  RD ,NOEXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)
;  $CODE$     76  NOVEC,NOWRT,  RD ,  EXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)


TESTFACT    11-Mar-1987 15:58:48    VAX BLISS--32 V4.3                  Page   3
            22-Jul-1983 16:06:56    DISK$:[DIRECTORY]MYPROG.B32;13          (1)

; Information:  0
; Warnings:     2
; Errors:       0


;                                       COMMAND QUALIFIERS

;       BLISS/LISTING MYPROG.B32

; Size:         76 code + 8 data bytes
; Run Time:        00:00.2
; Elapsed Time:    00:00.5
; Lines/CPU Min:    8250
; Lexemes/CPU-Min: 31000
; Memory Used:  19 pages
; Compilation Complete
```

G
Optional Programming Productivity Tools

This appendix provides an overview of optional programming productivity tools. These tools are not included with the BLISS–32 software; they must be purchased separately. Using these tools can increase your productivity as a BLISS–32 programmer. For information on how to purchase these tools, contact your DIGITAL sales representative.

G.1 Using LSE with BLISS–32

The VAX Language-Sensitive Editor (LSE) is a powerful and flexible text editor designed specifically for software development. LSE has important features that help you produce syntactically correct code in BLISS–32.

To invoke LSE, issue the LSEDIT command followed by a file name with a .B32 file type at the DCL prompt. For example:

```
$ LSEDIT USER.B32
```

The following sections describe some of the key features of LSE. Section G.1.1 discusses how to enter source code using LSE and Section G.1.2 describes LSE's compiler interface features. Section G.1.3 gives examples of how to generate BLISS–32 source code with LSE.

For more details on advanced features of LSE, see the *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer*.

G.1.1 Entering Source Code Using Tokens and Placeholders

LSE simplifies the tasks of developing and maintaining software systems. LSE provides the functions of a traditional text editor, plus additional powerful features: language-specific placeholders and tokens, aliases, comment and indentation control, and templates for subroutine libraries.

Placeholders are markers in the source code that indicate locations where you can provide program text. Placeholders help you to supply the appropriate syntax in a given context. Generally, you do not need to type placeholders; rather, they are inserted for you by LSE.

Placeholders are either optional or required. Required placeholders, which are delimited by braces ({}), represent places in the source code where you must provide program text. Optional placeholders, which are delimited by brackets ([]), represent places in the source code where you can either provide additional constructs or delete the placeholder.

There are three types of LSE placeholders:

Type of Placeholder	Description
Terminal	Provides text that describes valid replacements for the placeholder
Nonterminal	Expands into additional language constructs
Menu	Provides a list of options corresponding to the placeholder

You can move forward or backward from placeholder to placeholder. In addition, you can delete or expand placeholders as needed. Section G.1.3 shows examples of expanding placeholders.

Tokens typically represent keywords in BLISS–32. When expanded, tokens provide additional language constructs. You can type tokens directly into the buffer. Generally, you use tokens when you want to add language constructs and there are no placeholders in an existing program. For example, typing IF and issuing the EXPAND command causes a template for an IF construct to appear on your screen. You can also use tokens to by-pass long menus in cases where expanding a placeholder, such as {statement}, would result in a lengthy menu.

You can use tokens to insert text when editing an existing file by typing the name for a function or keyword and issuing the EXPAND command.

LSE commands allow you to manipulate tokens and placeholders. These commands and their default key bindings are as follows:

Command	Key Binding	Function
EXPAND	CTRL/E	Expands a placeholder
UNEXPAND	PF1-CTRL/E	Reverses the effect of the most recent placeholder expansion
GOTO PLACEHOLDER/FORWARD	CTRL/N	Moves the cursor to the next placeholder
GOTO PLACEHOLDER/REVERSE	CTRL/P	Moves the cursor to the previous placeholder
ERASE PLACEHOLDER/FORWARD	CTRL/K	Erases a placeholder
UNERASE PLACEHOLDER	PF1-CTRL/K	Restores the most recently erased placeholder
None	Down arrow	Moves the indicator down through a menu
None	Up arrow	Moves the indicator up through a menu
None	{ ENTER / RETURN }	Selects a menu option

You can display a list of all defined tokens and placeholders, or a particular token or placeholder, with the LSE commands SHOW TOKEN and SHOW PLACEHOLDER. To copy the listed information into a separate file, first issue the appropriate SHOW command to put the list into the $SHOW buffer. Then issue the following commands:

```
LSE> GOTO BUFFER $SHOW
LSE> WRITE filename
```

To obtain a hard copy of the list, use the PRINT command at DCL level to print the file you created.

G.1.2 Compiling Source Code

To compile your code and review compilation errors without leaving the editing session, you can use the LSE commands COMPILE and REVIEW. The COMPILE command issues a DCL command in a subprocess to invoke the BLISS–32 compiler. The compiler then generates a file of compile-time diagnostic information that LSE can use to review compilation errors. The diagnostic information is generated with the /DIAGNOSTICS qualifier that LSE appends to the compilation command.

For example, if you issue the COMPILE command while in the buffer USER.B32, the following DCL command executes:

```
$ BLISS USER.B32/DIAGNOSTICS=USER.DIA
```

LSE supports all of the BLISS–32 compiler's command qualifiers as well as user-supplied command procedures. You can specify DCL qualifiers, such as /LIBRARY, when invoking the compiler from LSE.

The REVIEW command displays any diagnostic messages that result from a compilation. LSE displays the compilation errors in one window and the corresponding source code in a second window so that you can review your errors while examining the associated source code. This capability eliminates tedious steps in the error correction process and helps ensure that all the errors are fixed before you recompile your program.

LSE provides several commands to help you review errors and examine your source code. These commands, and their default key bindings where applicable, are as follows:

Command	Key Binding	Function
COMPILE	None	Compiles the contents of the source buffer. You can issue this command with the /REVIEW qualifier to put LSE in REVIEW mode immediately after the compilation.
REVIEW	None	Puts LSE into REVIEW mode and displays any errors resulting from the last compilation.
END REVIEW	None	Removes the buffer $REVIEW from the screen; returns the cursor to a single window containing the source buffer.
GOTO SOURCE	CTRL/G	Moves the cursor to the source buffer that contains the error.
NEXT STEP	CTRL/F	Moves the cursor to the next error in the buffer $REVIEW.
PREVIOUS STEP	CTRL/B	Moves the cursor to the previous error in the buffer $REVIEW.
	{ Down arrow Up arrow }	Moves the cursor within a buffer.

G.1.3 Examples

The following sections show examples of using some common tokens and placeholders to write BLISS–32 code. The examples are expanded to show the formats and guidelines LSE provides; however, not all of the examples are fully expanded.

The examples show expansions of the following BLISS–32 features:

- Module declaration

- Routine declaration
- IF statement
- Select expression
- CH$COMPARE function

Instructions and explanations precede each example, and an arrow (→) indicates the line in the code where an action has occurred.

See Section G.1.1 for the commands that manipulate tokens and placeholders.

Remember that braces and tildes (~{ }~) enclose required placeholders; brackets and tildes (~ [ ] ~) enclose optional placeholders. Note that when you erase an optional placeholder, LSE also deletes any associated text before and after that placeholder.

Note

Keywords such as MODULE, ROUTINE, IF, SELECT, and CH$COMPARE can be tokens as well as placeholders; therefore, any time you are in the BLISS–32 language environment you can type one of these words and press CTRL/E to expand the construct.

When you use LSE to create a new BLISS–32 program, the initial string [~ MODULE ~] appears at the top of the screen.

G.1.4 Module Declaration

Expand the token [~ MODULE ~].

```
[~%TITLE '[~quoted_chars~]'~]
MODULE {~name~} [~(module_switches)~] =
BEGIN
[~software_copyright_statement~]
![~module_level_comments~]
[~module_level_declarations~]
[~routine_declaration~]...
END                          ! End of module
ELUDOM
```

Erase the optional placeholder [~%TITLE '[~quoted_chars~]'~] and expand the placeholder [~(module_switches)~] twice to display a menu. Select the option {~special_switch~} to display another menu, and select the option IDENT.

```
-> MODULE {~name~} (IDENT = '[~quoted_chars~]' , [~module_switches~]...) =
   BEGIN
   [~software_copyright_statement~]
   ![~module_level_comments~]
   [~module_level_declarations~]
   [~routine_declaration~]...
   END                          ! End of module
   ELUDOM
```

Expand the placeholder [~module_level_comments~].

```
MODULE {~name~} (IDENT = '[~quoted_chars~]' , [~module_switches~]...) =
[~software_copyright_statement~]
BEGIN
!++
! FACILITY:
!
!   [~tbs~]
!
! ABSTRACT:
!
!   [~tbs~]
!
! AUTHORS:
!
!   [~tbs~]
!
!
! CREATION DATE: [~tbs~]
!
! MODIFICATION HISTORY:
!--
[~module_level_declarations~]
[~routine_declaration~]...
END                           ! End of module
ELUDOM
```

G.1.4.1 Routine Declaration

Expand the placeholder [~routine_declaration~] to display a menu.

```
->       [~routine_declaration~]
```

Select the option COMPLEX_ROUTINE.

```
   [~%SBTTL '[~quoted_chars~]'~]
-> [~GLOBAL~] ROUTINE {~name~} [~(formals)~] : [~routine_attributes~] =
   [~routine_level_comments~]
       BEGIN
       [~declaration~]...
       {~expression~}...
       END;
   [~routine_declaration~]...
```

Erase the placeholder [~GLOBAL~], expand [~(formals)~], and erase [~routine_attributes~].

```
   [~%SBTTL '[~quoted_chars~]'~]
-> ROUTINE {~name~} ([~input_formals~] [~; output_formals~]) =
   [~routine_level_comments~]
       BEGIN
       [~declaration~]...
       {~expression~}...
       END;
   [routine_declaration~]...
```

Erase the placeholder [~input_formals~] and expand [~output_formals~].

```
   [~%SBTTL '[~quoted_chars~]'~]
-> ROUTINE {~name~} ( ; {~formal_items~}...) =
   [~routine_level_comments~]
       BEGIN
       [~declaration~]...
       {~expression~}...
       END;
   [~routine_declaration~]...
```

Expand the placeholder {~formal_items~}.

```
   [~%SBTTL '[~quoted_chars~]'~]
-> ROUTINE {~name~} (; {~name~} : [~formal_attributes~]..., [~formal_items~]...) =
   [~routine_level_comments~]
       BEGIN
       [~declaration~]...
       {~expression~}...
       END;
   [~routine_declaration~]...
```

Type Outarg over the second {~name~} placeholder and expand the placeholder [~formal_attributes~].

```
   [~%SBTTL '[~quoted_chars~]'~]
-> ROUTINE {~name~} (; Outarg : {~map_attributes~} [~formal_attributes~] ,
           [~formal_items~]...)=
   [~routine_level_comments~]
       BEGIN
       [~declaration~]...
       {~expression~}...
       END;
   [~routine_declaration~]...
```

Expand the placeholder {~map_attributes~} to display a menu. Select the option {~structure_attribute~} to display a menu. Select the option REF {~structure_type~}. Expand the placeholder {~structure_type~} to display a menu. Select REF_VECTOR, and erase the optional placeholder [{~number_of_elements~}[~allocation_and_or_extension~]] and the optional list placeholders [~formal_attributes~] and [~formal_items~] . . . .

```
   [~%SBTTL '[~quoted_chars~]'~]
-> ROUTINE {~name~} (; Outarg : REF VECTOR[]) =
   [~routine_level_comments~]
       BEGIN
       [~declaration~]...
       {~expression~}...
       END;
   [~routine_declaration~]...
```

G.1.4.2 IF Statement

Expand the placeholder {~expression~} to display a menu.

```
   BEGIN
   [~declaration~]...
-> {~expression~}...
   END;
```

Select the option IF.

```
   BEGIN
   [~declaration~]...
-> IF {~expression~}
   THEN
           {~expression~}
   [~ELSE  {~expression~} ~];
   [~expression~]...
   END;
```

Expand the placeholder [~ELSE {~expression~} ~].

```
   BEGIN
   [~declaration~]...
   IF {~expression~}
   THEN
        {~expression~}
-> ELSE
        {~expression~};
   [~expression~]...
   END;
```

G.1.4.3 Select Expression

Expand the placeholder {~expression~} to display a menu.

```
   BEGIN
   [~declaration~]...
-> {~expression~}...
   END;
```

Select the option SELECT.

```
    BEGIN
    [~declaration~]...
->  SELECT[~select_type~] {~select_index~} OF
        SET
        [{~select_labels~}...]:
            {~select_action~};
        {~sel_lines~}...
        TES;
    [~expression~]...
    END;
```

Expand the placeholder [~select_type~] to display a menu and select the option ONE. Type .value over the placeholder {~select_index~}.

```
    BEGIN
    [~declaration~]...
->  SELECTONE .value OF
        SET
        [{~select_labels~}...]:
            {~select_action~};
        {~sel_lines~}...
        TES;
    [~expression~]...
    END;
```

Expand the placeholder {~select_labels~} . . . to display a menu and select the option {~low_selector~} TO {~high_selector~}.

```
    BEGIN
    [~declaration~]...
    SELECTONE .value OF
        SET
->      [{~low_selector~} TO {~high_selector~}, [~select_labels~]...]:
            {~select_action~};
        {~sel_lines~}...
        TES;
    [~expression~]...
    END;
```

Type the value –128 over the placeholder {~low_selector~} and the value 127 over the placeholder {~high_selector~}. Erase the remaining duplicated placeholder [select_labels] . . . .

```
    BEGIN
    [~declaration~]...
    SELECTONE .value OF
        SET
->      [-128 TO 127]:
            {~select_action~};
        {~sel_lines~}...
        TES;
    [~expression~]...
    END;
```

Expand the placeholder {~select_action~} to produce the placeholder {~expression~} and type size = 1 over {~expression~}.

```
    BEGIN
    [~declaration~]...
    SELECTONE .value OF
        SET
        [-128 TO 127]:
->          size = 1;
        [~sel_lines~]...
        TES;
    [~expression~]...
    END;
```

Expand the placeholder {~sel_lines~} . . . .

```
    BEGIN
    [~declaration~]...
    SELECTONE .value OF
        SET
        [-128 TO 127]:
            size = 1;

->      [{~select_labels~}...]:
            {~select_action~};
        [~sel_lines~]...
        TES;
    [~expression~]...
    END;
```

Expand the placeholder {~select_labels~} to display a menu and select the option OTHERWISE.

```
    BEGIN
    [~declaration~]...
    SELECTONE .value OF
        SET
        [-128 TO 127]:
            size = 1;

->      [OTHERWISE, [~select_labels~]...]:
            {~select_action~};
        [~sel_lines~]...
        TES;
    [~expression~]...
    END;
```

Erase the optional list placeholder [~select_labels~] . . . . Type size = 4 over the placeholder {~select_action~}. Erase the optional list placeholder [~sel_lines~].

```
    BEGIN
    [~declaration~]...
    SELECTONE .value OF
        SET
        [-128 TO 127]:
        size = 1;

        [OTHERWISE]:
->          size = 4;
        TES;
    [~expression~]...
    END;
```

G.1.4.4 CH$COMPARE Function

```
    BEGIN
    [~declaration~]...
    {~expression~}...
    END;
```

Type CH$COMPARE over the placeholder {~expression~} . . . and expand CH$COMPARE.

```
    BEGIN
    [~declaration~]...
->  CH$COMPARE ({~n1_ch~}, {~ptr_ch~}, {~n2_ch~}, {~ptr2_ch~}, [~fill_ch~]);
    [~expression~]...
    END;
```

Type the value 5 over the placeholder {~n1_ch~} and CH$PTR (alpha) over the placeholder {~ptr1_ch~}.

```
    BEGIN
    [~declaration~]...
->  CH$COMPARE (5, CH$PTR (alpha), {~n2_ch~}, {~ptr2_ch~}, [~fill_ch~]);
    [~expression~]...
    END;
```

Type the value 7 over the placeholder {~n2_ch~} and CH$PTR (beta) over the placeholder {~ptr2_ch~}.

```
    BEGIN
    [~declaration~]...
->  CH$COMPARE (5, CH$PTR (alpha), 7, CH$PTR (beta), [~fill_ch~]);
    [~expression~]...
    END;
```

Expand the placeholder [~fill_ch~].

```
    BEGIN
    [~declaration~]...
->  CH$COMPARE (5, CH$PTR (alpha), 7, CH$PTR (beta), {~character_code_literal~});
    [~expression~]...
    END;
```

Expand the placeholder {~character_code_literal~}.

```
    BEGIN
    [~declaration~]...
->  CH$COMPARE (5, CH$PTR (alpha), 7, CH$PTR (beta), %C'{~quoted_character~}');
    [~expression~]...
    END;
```

Type a blank over the placeholder {~quoted_character~}.

```
    BEGIN
    [~declaration~]...
->  CH$COMPARE (5, CH$PTR (alpha), 7, CH$PTR (beta), %C' ');
    [~expression~]...
    END
```

G.2 Using the VAX Source Code Analyzer

The VAX Source Code Analyzer (SCA) is an interactive tool used to cross-reference and analyze source code. It can be used with most VAX programming languages. SCA helps developers monitor large, complex software systems by displaying source information in response to user queries. SCA stores data generated by the BLISS–32 compiler in an SCA library. The data in an SCA library contains information about all the symbols, modules, and files encountered during a specific compilation of the source.

SCA lets you query both cross-reference and static analysis information. Cross-referencing supplies information about program symbols and source files. SCA provides the following cross-referencing features:

- Locating names and occurrences (uses) of these names
- Querying several names or partial names (with wildcards allowed)
- Limiting a query to specific characteristics (such as routine names, variable names, or source files)
- Limiting a query to specific occurrences (such as the primary declaration of a symbol, read or write occurrences of a symbol, or occurrences of a file)

The static analysis query features of SCA provide structural information on the interrelation of routines, symbols, and files. SCA provides the following static analysis features:

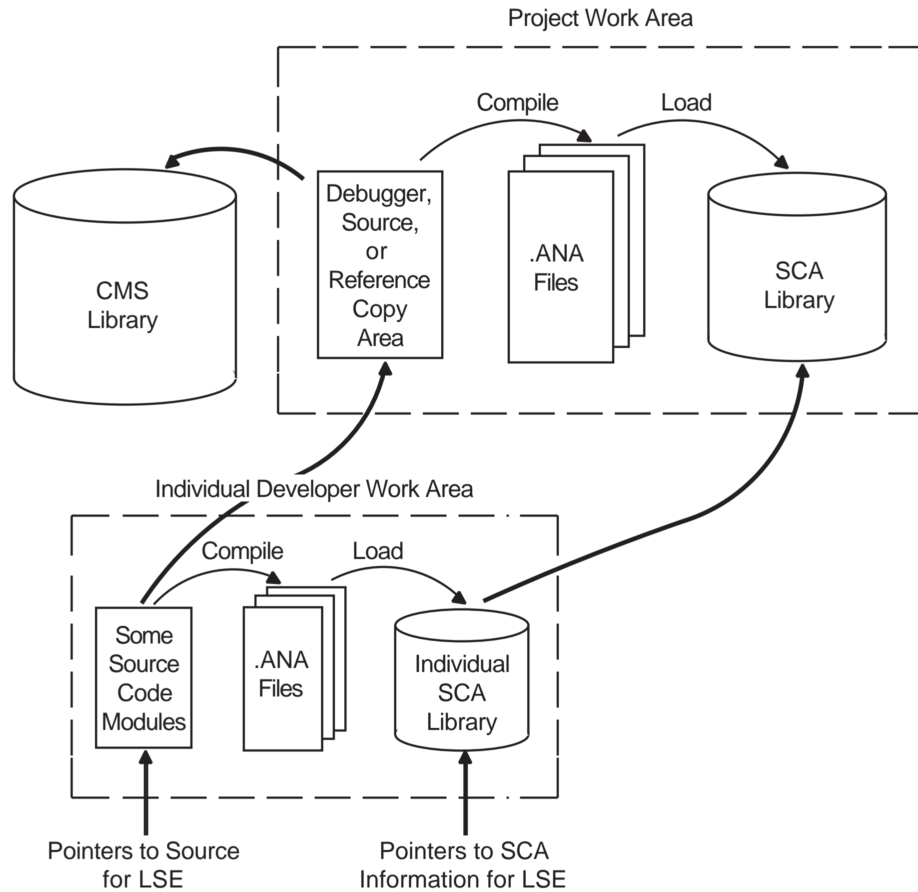
- Displaying routine calls to and from a specified routine
- Analyzing routine calls for consistency as to the numbers and data types of arguments passed, and the types of values returned

SCA is fully integrated with LSE to provide additional features. By using SCA with LSE, you can view any portion of an entire system and edit related source files.

Multimodular Development

The cross-referencing and static analysis features of SCA are especially useful during the implementation and maintenance phases of a project that involves many programming modules. For example, the project team work area in Figure G–1 contains a set of source modules. (The team might use a code management tool, such as VAX DEC/CMS, to keep track of these modules in their various development stages.) When the team compiles the source code, SCA generates the source information it requires (that is, data analysis files with the file type .ANA); then the team loads this information into a previously established project SCA library.

Figure G–1 Use of LSE and SCA for Multimodular Development



When a team member wants to do additional development work on specific modules, that member sets up an individual work area, which might contain the following:

- Copies of source and object modules from the project libraries
- Local SCA libraries that contain copies of the module information

To make available all the capabilities of SCA/LSE integration, the team member informs LSE of the locations of that member's current sources and related source information. Using LSE, all team members can effectively "see through" their own individual work areas to the project work area and possibly to other individual work areas.

The following sections provide a general overview of SCA and discuss some of the commands that are available to you when you use SCA within LSE. For detailed information on SCA, refer to the *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer*.

G.2.1 Setting Up an SCA Environment

If you do not have an existing SCA library, you must take the following steps to set up an SCA environment:

1. Create an SCA library in a subdirectory.
2. Use the BLISS–32 compiler to generate the data analysis (.ANA) files for each source module in your system.
3. Load these data analysis files into your local SCA library.

If you have an existing SCA library, you only need to select the library to use SCA to conduct source information queries.

G.2.1.1 Creating an SCA Library

To create an SCA library, first create a subdirectory at the DCL level. For example:

```
$ CREATE/DIRECTORY PROJ:[USER.LIB1]
```

This command creates a subdirectory LIB1 for a local SCA library.

To initialize a new SCA library, issue the command CREATE LIBRARY.

For example:

```
$ SCA CREATE LIBRARY [.LIB1]
```

This command initializes and activates library LIB1.

G.2.1.2 Generating Data Analysis Files

SCA uses detailed source data that is generated by the BLISS–32 compiler. When you specify the /ANALYSIS_DATA qualifier on the BLISS command, the generated data is output to a file with the default type .ANA. For example:

```
$ BLISS/LIST/DIAGNOSTICS/ANALYSIS_DATA PG1,PG2,PG3
```

This command line compiles the input files PG1.B32, PG2.B32, and PG3.B32, and generates four corresponding output files for each input file, with the file types .OBJ, .LIS, .DIA and .ANA. SCA puts these files in your current default directory unless you specify otherwise.

G.2.1.3 Loading Data Analysis Files into a Local Library

Before you can examine the information in the .ANA files, you must load the information into an SCA library using the LOAD command.

For example:

```
LSE> LOAD PG1,PG2,PG3
```

This command loads your library with the modules contained in the data analysis files PG1.ANA, PG2.ANA, and PG3.ANA.

G.2.1.4 Selecting an SCA Library

To select an existing SCA library to use with your current SCA session, use the SCA command SET LIBRARY.

For example:

```
LSE> SET LIBRARY [.LIB]
```

This command activates library LIB1.

G.2.2 Using SCA for Cross-Referencing

Once you have set up your SCA environment, you can ask for symbol or file information by using the SCA command FIND. The FIND command has the following form:

```
FIND [/qualifier...] [name-expression[,...]]
```

The name expression can be explicit or can contain wildcards. For example:

```
LSE> FIND ABC,XY%
```

You can query an SCA library for the following:

Name	A series of characters that uniquely identifies a symbol or a file
Item	An appearance of a symbol (such as a variable, constant, label, or procedure) or a file
Occurrence	The use of a symbol or a file

To limit the information resulting from a query, you can use qualifiers on the FIND command. For example:

```
LSE> FIND/REFERENCES=CALL BUILD_TABLE
```

This command causes SCA to report only references in the source code where the routine BUILD_TABLE is called.

When you first issue a FIND command within LSE, you initiate a query session. Within this context, the integration of LSE and SCA provides commands that can be used only within LSE. These commands are as follows:

Command	Function
{ NEXT | PREVIOUS } { NAME | ITEM | OCCURRENCE | QUERY | STEP }	Lets you step through one or more query buffer displays within LSE
GOTO SOURCE	Displays the source corresponding to the current query item
GOTO DECLARATION	Positions the cursor on a symbol declaration in one window, and displays the source code that contains the symbol declaration in another window

Index

A

Abbreviations
- qualifiers, 1–18, A–3
- qualifier values, A–3
- value names, 1–18

ADAWI function, 4–6
ADDD function, 4–7
ADDF function, 4–7
ADDG function, 4–7
ADDH function, 4–8
ADDM function, 4–8
/ANALYSIS_DATA qualifier, 1–13, A–3
%ASCII literal, 6–4
ASHP function, 4–8
ASHQ function, 4–9
ASSEMBLER qualifier value, 1–9

B

BICPSW function, 4–10
BINARY qualifier value, 1–9
BIND declaration, 5–1
BIND ROUTINE declaration, 5–1
BISPSW function, 4–10
BLISS–16, 6–1
%BLISS16 lexical, 6–5
%BLISS32 lexical, 6–5
BLISS–36, 6–1
%BLISS36 lexical, 6–5
BLISS module template, C–1
- closing format, C–3
- declararative part, C–2
- executable part, C–2
- preface, C–1

%BPADDR literal, 6–3
BPT function, 4–10
%BPUNIT literal, 6–3
%BPVAL literal, 6–3
BUGL function, 4–10
BUGW function, 4–11
BUILTIN declaration, 5–1

C

CALLG function, 4–11
Characters, 1–2
/CHECK qualifier, 1–5, A–3
/CHECK values
- FIELD, A–3
- INITIAL, A–3
- OPTIMIZE, A–3
- REDECLARE, A–3

CHME function, 4–12
CHMK function, 4–12
CHMS function, 4–12
CHMU function, 4–12
%C literal, 6–4
CMPC3 function, 4–12
CMPC5 function, 4–13
CMPD function, 4–14
CMPF function, 4–14
CMPG function, 4–14
CMPH function, 4–15
CMPM function, 4–15
CMPP function, 4–16
Code examples, 9–1
Code formatting, B–1
/CODE qualifier, 1–8, A–3
CODE switch, 1–17
Coding errors, 5–2
- block values, 5–3
- from linker, 5–7
- indexed loop, 5–6
- macro usage, 5–5
- missing code, 5–5
- missing dots, 5–3
- nonvalued routines, 5–3
- operator precedence, 5–4
- PC usage, 5–5
- routine calls, 5–4
- semicolons, 5–3
- signed fields, 5–4
- unsigned fields, 5–4
- valued routines, 5–3

Coding examples
- ALPHA program, 9–1
- CALC program, 9–10

Command line
  semantics, 1–2
  syntax, 1–2
    summary, A–1
COMMENTARY qualifier value, 1–9
Compilation
  summary, 2–2
Compiler, 7–1
  optimization, 7–1
  output, 2–1
  overview, 7–1
  phases, 7–1
    CODE, 7–1
    code generation, 7–7
    code optimization, 7–7
    DELAY, 7–1
    FINAL, 7–1
    FLOW, 7–1
    flow analysis, 7–2
    heuristic, 7–6
    lexical analysis, 7–1
    LEXSYN, 7–1
    name binding, 7–6
    OUTPUT, 7–1
    output files, 7–7
    syntactic analysis, 7–1
    TNBIND, 7–1
  switch effects, 7–7
  switches, 7–1
    CODE, 7–1
    NOCODE, 7–1
    NOOPTIMIZE, 7–1
    NOSAFE, 7–1
    NOZIP, 7–1
    OPTIMIZE, 7–1
    OPTLEVEL, 7–1
    SAFE, 7–1
    ZIP, 7–1
COMPILETIME declaration, 5–1
Concatenation, 1–2
  indicator, 1–2
CRC function, 4–16
Cross-reference listing, 2–19
  defined field, 2–21
  header, 2–19
  referenced field, 2–22
  symbol field, 2–19
  type field, 2–19
/CROSS_REFERENCE qualifier, 1–6, A–4
/CROSS_REFERENCE value
  MULTIPLE, A–4
  NOMULTIPLE, A–4
CVTDF function, 4–17
CVTDI function, 4–17
CVTDL function, 4–17
CVTFD function, 4–18
CVTFG function, 4–18
CVTFH function, 4–18
CVTFI function, 4–19
CVTFL function, 4–19
CVTGF function, 4–19
CVTGH function, 4–20
CVTGL function, 4–20
CVTHF function, 4–20
CVTHG function, 4–20
CVTHL function, 4–21
CVTID function, 4–21
CVTIF function, 4–21
CVTLD function, 4–22
CVTLF function, 4–22
CVTLH function, 4–22
CVTLP function, 4–23
CVTPL function, 4–23
CVTPS function, 4–23
CVTPT function, 4–24
CVTRDL function, 4–25
CVTRFL function, 4–25
CVTRGL function, 4–25
CVTRHL function, 4–26
CVTSP function, 4–26
CVTTP function, 4–27

D

Debugging, 3–2
  address expressions, 3–5
  arithmetic operators, 3–4
  commands
    summary, 3–13
    syntax, 3–3
  /DEBUG, 3–13
  display mode, 3–3
  entry mode, 3–3
  field references, 3–8
  HEXIDECIMAL, 3–3
  REF structure reference, 3–11
  scope of names, 3–12
  source-line, 3–12
  special characters, 3–5
  structure references, 3–9
  SYMBOLIC, 3–3
/DEBUG qualifier, 1–8, 3–13, A–4
DEBUG switch, 1–17
Declarations
  BIND, 5–1
  BIND ROUTINE, 5–1
  BUILTIN, 5–1
  COMPILETIME, 5–1
  EXTERNAL, 5–1
  EXTERNAL LITERAL, 5–1
  EXTERNAL ROUTINE, 5–1
  FIELD, 5–1
  GLOBAL, 6–10
  KEYWORDMACRO, 5–1

Declarations (cont'd)
LIBRARY, 5–1
LINKAGE, 5–1
LITERAL, 5–1, 6–4
LOCAL, 6–10
MACRO, 5–1
OWN, 6–10
REQUIRE, 5–1
STRUCTURE, 5–1
SWITCHES, 5–1
UNDECLARE, 5–1
Defaults
qualifiers, A–3
Delimiters, 1–2
DIVD function, 4–27
DIVF function, 4–28
DIVG function, 4–28
DIVH function, 4–28

E

Editors
LSE, G–1 to G–13
EDITPC function, 4–29
EDIV function, 4–30
%ELSE lexical, 6–5
EMUL function, 4–30
Error messages, E–1
format of, 2–27
pointers, 2–29
types of, 2–27
ERRORS qualifier value, 1–16
/ERROR_LIMIT qualifier, 1–8, A–4
ERRS switch, 1–17
%EXPAND, 5–7
EXPAND_MACROS qualifier value, 1–15
EXTERNAL declaration, 5–1
EXTERNAL LITERAL declaration, 5–1
EXTERNAL ROUTINE declaration, 5–1

F

FFC function, 4–30
FFS function, 4–30
FIELD declaration, 5–1
FIELD qualifier value, 1–5
File-designator, 1–4, 1–13
Files
LIBRARY, 5–1
REQUIRE, 5–1
File type
defaults, 1–4
%FI lexical, 6–5
Format
error messages, 2–27
preface string, 2–5

G

GLOBAL declaration, 6–10

H

HALT function, 4–31
HEADER qualifier value, 1–15
HEXIDECIMAL, 3–3

I

%IF lexical, 6–5
Implementation limits, D–1
BLISS–32 constructs, D–1
System interfaces, D–1
INDEX function, 4–31
INITIAL qualifier value, 1–5
INSQHI function, 4–32
INSQTI function, 4–32
INSQUE function, 4–32

K

KEYWORDMACRO declaration, 5–1

L

Language-Sensitive Editor (LSE), 8–1, G–1 to G–13
LEVEL qualifier value, 1–11
Lexical functions
%BLISS16, 6–5
%BLISS32, 6–5
%BLISS36, 6–5
%ELSE, 6–5
%EXPAND, 5–7
%FI, 6–5
%IF, 6–5
%QUOTE, 5–7
%REMAINING, 6–5
%THEN, 6–5
Libraries, 8–1
LIBRARY
declaration, 5–1
files, 5–1, 6–8
/LIBRARY qualifier, 1–13, A–4
LIBRARY qualifier value, 1–15
LINKAGE declaration, 5–1
Linking, 3–1
commands, 3–1
example, 3–2
/LIST qualifier, 1–13, A–4
LIST switch
ASSEMBLY, 1–17
BINARY, 1–17
COMMENTARY, 1–17

LIST switch (cont'd)
EXPAND, 1–17
LIBRARY, 1–17
OBJECT, 1–17
REQUIRE, 1–17
SOURCE, 1–17
SYMBOLIC, 1–17
TRACE, 1–17
LITERAL declaration, 5–1, 6–4
LOCAL declaration, 6–10
LOCC function, 4–33
LSE (Language-Sensitive Editor), 8–1, G–1 to G–13

M

Machine-specific functions, 4–1
input parameters, 4–1
instruction summary, 4–2
output parameters, 4–2
/MACHINE_CODE_LIST qualifier, 1–9, A–4
/MACHINE_CODE_LIST values
ASSEMBLER, A–4
BINARY, A–4
COMMENTARY, A–4
OBJECT, A–4
SYMBOLIC, A–4
UNIQUE_NAMES, A–4
MACRO declaration, 5–1
Macros, 5–7
conditional, 5–7
creating a set, 5–9
enumeration types, 5–9
iterative, 5–7
lexical functions, 5–7
machine dependencies, 5–7
set elements, 5–11
set memberships, 5–12
SET type, 5–9
string instructions, 5–8
MATCHC function, 4–33
MFPR function, 4–34
Module-head switch, 1–17
values
CODE, 1–17
DEBUG, 1–17
ERRS, 1–17
LIST
ASSEMBLY, 1–17
BINARY, 1–17
COMMENTARY, 1–17
EXPAND, 1–17
LIBRARY, 1–17
OBJECT, 1–17
REQUIRE, 1–17
SOURCE, 1–17
SYMBOLIC, 1–17
TRACE, 1–17
Module-head switch
values (cont'd)
NOZIP, 1–17
OPTLEVEL, 1–17
SAFE, 1–17
UNAMES, 1–17
ZIP, 1–17
MOVC3 function, 4–34
MOVC5 function, 4–35
MOVP function, 4–36
MOVPSL function, 4–36
MOVTC function, 4–37
MOVTUC function, 4–37
MTPR function, 4–38
MULD function, 4–39
MULF function, 4–39
MULG function, 4–39
MULH function, 4–40
MULTIPLE qualifier value, 1–7

N

NOP function, 4–40
NOZIP switch, 1–17

O

Object listing
default, 2–8
/OBJECT qualifier, 1–13, A–4
Operating procedures
compiling, 1–1
OPLEVEL switch, 1–17
/OPTIMIZE qualifier, 1–11, A–4
OPTIMIZE qualifier value, 1–5
/OPTIMIZE values
HEADER, A–4
LEVEL, A–4
LIBRARY, A–4
PAGE_SIZE, A–4
QUICK, A–4
REQUIRE, A–4
SAFE, A–4
SOURCE, A–4
SPACE, A–4
SPEED, A–4
TRACE_MACROS, A–4
Output listing, 2–3
header, 2–4
object part, 2–7
source part, 2–5
OWN declaration, 6–10

P

PAGE_SIZE qualifier value, 1–15
PIC, 5–7
PLIT, 6–16
  defined, 6–17
  scalar, 6–17
  strings, 6–17
Position-independent code, 5–7
Predeclared literals
  %ASCII, 6–4
  %BPADDR, 6–3
  %BPUNIT, 6–3
  %BPVAL, 6–3
  %C, 6–4
  %UPVAL, 6–3
PROBER function, 4–40
PROBEW function, 4–41

Q

Qualifiers
  abbreviations, 1–18, A–3
  /ANALYSIS_DATA, 1–13
  /CODE, 1–8
  /CROSS_REFERENCE, 1–6
  /DEBUG, 1–8, 3–13
  defaults, A–3
  default settings, 1–17
  /ERROR_LIMIT, 1–8
  /LIBRARY, 1–13
  /LIST, 1–13
  /MACHINE_CODE_LIST, 1–9
  negative forms, 1–18
  /OBJECT, 1–13
  positive forms, 1–18
  /QUICK, 1–11
  /SOURCE_LIST, 1–14
  /TERMINAL, 1–16
  /TRACEBACK, 1–8
  /VARIANT, 1–8
Qualifiers>/CHECK, 1–5
Qualifiers>/OPTIMIZE, 1–11
/QUICK qualifier, 1–11
QUICK qualifier value, 1–11
%QUOTE, 5–7

R

REDECLARE qualifier value, 1–5
%REMAINING lexical, 6–5
REMQHI function, 4–41
REMQTI function, 4–41
REMQUE function, 4–42
REQUIRE
  declaration, 5–1
  files, 5–1, 6–8
REQUIRE qualifier value, 1–15
ROT function, 4–42

S

SAFE qualifier value, 1–11
SAFE switch, 1–17
Sample output listing, F–1
SCA (Source Code Analyzer), 8–1, G–10 to G–13
SCANC function, 4–43
SKPC function, 4–43
Source Code Analyzer (SCA), 8–1, G–10 to G–13
/SOURCE_LIST qualifier, 1–14, A–4
/SOURCE_LIST values
  EXPAND_MACROS, A–4
SPACE qualifier value, 1–11
SPANC function, 4–44
SPEED qualifier value, 1–11
STATISTICS qualifier value, 1–16
STRUCTURE declaration, 5–1
SUBD function, 4–44
SUBF function, 4–45
SUBG function, 4–45
SUBH function, 4–45
SUBM function, 4–46
Switches
  CODE, 1–17
  DEBUG, 1–17
  ERRS, 1–17
  LIST, 1–17
  NOZIP, 1–17
  OPLEVEL, 1–17
  SAFE, 1–17
  UNAMES, 1–17
  ZIP, 1–17
SWITCHES declaration, 1–17, 5–1
  values
    ERRS, 1–17
    LIST
      ASSEMBLY, 1–17
      BINARY, 1–17
      COMMENTARY, 1–17
      EXPAND, 1–17
      LIBRARY, 1–17
      OBJECT, 1–17
      REQUIRE, 1–17
      SOURCE, 1–17
      SYMBOLIC, 1–17
      TRACE, 1–17
    NOZIP, 1–17
    SAFE, 1–17
    UNAMES, 1–17
    ZIP, 1–17
SWITCHES-declaration, 1–17
SYMBOLIC, 3–3
SYMBOLIC qualifier value, 1–9

System services, 8–1
  routine interfaces, 8–2
  sample program, 8–2

T

Terminal output, 2–2
/TERMINAL qualifier, 1–16, A–4
/TERMINAL values
  ERRORS, A–4
  STATISTICS, A–4
TESTBITCC function, 4–46
TESTBITCCI function, 4–46
TESTBITCS function, 4–46
TESTBITSC function, 4–46
TESTBITSS function, 4–46
TESTBITSSI function, 4–46
%THEN lexical, 6–5
Tools, 8–1
  Tutorial Terminal I/O, 8–1
  VAX Language-Sensitive Editor, 8–1, G–1 to G–13
  VAX Source Code Analyzer, 8–1, G–10 to G–13
/TRACEBACK qualifier, 1–8, A–4
TRACE_MACROS qualifier value, 1–15
Transportability, 6–1
  concepts, 6–1
  guidelines, 6–1
  parallel definitions, 6–1
  parametization, 6–1
  strategies, 6–2
    isolation, 6–2
    simplicity, 6–3
  techniques, 6–10
    addresses, 6–12
    data, 6–10
    field selectors, 6–25
    initialization, 6–19
    PLIT, 6–16
    strings, 6–15
    structures, 6–25
  tools, 6–3
    library files, 6–8
    literals, 6–3
    macros, 6–5
    names, 6–7
    REQUIRE files, 6–8
    routines, 6–9
    switches, 6–5
Tutorial Terminal I/O (TUTIO), 8–1

U

UNAMES switch, 1–17
UNDECLARE declaration, 5–1
UNIQUE_NAMES qualifier value, 1–9
%UPVAL literal, 6–3

V

Values
  abbreviations, 1–18, A–3
  /CHECK qualifier
    FIELD, 1–5
    INITIAL, 1–5
    OPTIMIZE, 1–5
    REDECLARE, 1–5
  /CROSS_REFERENCE qualifier
    MULTIPLE, 1–7
  /MACHINE_CODE_LIST qualifier
    ASSEMBLER, 1–9
    BINARY, 1–9
    COMMENTARY, 1–9
    SYMBOLIC, 1–9
    UNIQUE_NAMES, 1–9
  /OPTIMIZE qualifier
    LEVEL, 1–11
    QUICK, 1–11
    SAFE, 1–11
    SPACE, 1–11
    SPEED, 1–11
  /SOURCE_LIST qualifier
    EXPAND_MACROS, 1–15
    HEADER, 1–15
    LIBRARY, 1–15
    PAGE_SIZE, 1–15
    REQUIRE, 1–15
    TRACE_MACROS, 1–15
  /TERMINAL qualifier
    ERRORS, 1–16
    STATISTICS, 1–16
/VARIANT qualifier, 1–8, A–4

X

XFC function, 4–47

Z

ZIP switch, 1–17