BLISS–36
User's Guide

Order No. AA-H712D-TK

February 1984

This document describes the BLISS-36 compiler and its use, gives basic information about linking, executing, and debugging BLISS-36 programs, and describes BLISS-36 machine-specific functions, BLISS tools, and other topics relevant to BLISS-36 programming.

SUPERSESSION/UPDATE INFORMATION: BLISS-36 V4(216)

OPERATING SYSTEMS AND VERSIONS: TOPS-10 V7.01A
TOPS-20 V5.1(KL)
TOPS-20 V4.1(KS)

SOFTWARE VERSION: BLISS-36 V4(216) implementing BLISS language V4.0

digital equipment corporation · maynard, massachusetts

First Printing, June 1979
Revised, September 1980
Revised, February 1982
Revised, February 1984

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

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

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

Printed in U.S.A.

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	RSTS
DEC/CMS	EduSystem	RSX
DEC/MMS	IAS	UNIBUS
DECnet	MASSBUS	VAX
DECsystem-10	MICRO/PDP-11	VMS
DECSYSTEM-20	Micro/RSX	VT
DECUS	PDP	digital
DECwriter	PDT	

ZK2260

HOW TO ORDER ADDITIONAL DOCUMENTATION

In Continental USA and Puerto Rico call 800-258-1710

In New Hampshire, Alaska, and Hawaii call 603-884-6660

In Canada call 613-234-7726 (Ottawa-Hull)
800-267-6146 (all other Canadian)

DIRECT MAIL ORDERS (USA & PUERTO RICO)*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire 03061

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575)

DIRECT MAIL ORDERS (CANADA)

Digital Equipment of Canada Ltd.
940 Belfast Road
Ottawa, Ontario K1G 4C2
Attn: A&SG Business Manager

DIRECT MAIL ORDERS (INTERNATIONAL)

Digital Equipment Corporation
A&SG Business Manager
c/o Digital's local subsidiary or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

CONTENTS

PREFACE

MANUAL OBJECTIVES

This manual is a user's guide for the BLISS-36 compiler, which runs on TOPS-10 and TOPS-20 operating systems. It provides three kinds of information: basic operating instructions, advanced material, and reference information. This manual is intended as a companion manual to the BLISS Language Guide. As such, they have certain structural similarities, and the discussions of organization and syntax notation given in the language guide apply to this manual.

INTENDED AUDIENCE

This guide is intended for users of the BLISS-36 programming language. It presupposes some familiarity with the TOPS-10 or TOPS-20 operating system, its command language, and file-system conventions.

STRUCTURE OF THIS DOCUMENT

Chapters 1 through 4 describe basic operating instructions:

- Chapters 1 and 2 present procedures for compiling a BLISS program, define file specifications, and describe command switches that can be used in the TOPS-20 and TOPS-10 operating system environments.
- Chapter 3 considers output produced by the compilation. The format and meaning of each of the possible compiler outputs are described and illustrated.
- Chapter 4 is concerned with linking, executing, and debugging.

Chapters 5 through 10 supply advanced material:

- Chapter 5 describes machine-specific functions.
- Chapter 6 describes programming considerations, such as the use of LIBRARY and REQUIRE facilities.
- Chapter 7 gives guidelines for writing transportable BLISS programs.
- Chapter 8 presents a discussion of the compiler architecture and provides insight into the effects that result from command switches related to optimization.

- Chapter 9 describes some tools related to BLISS programming.
- Chapter 10 provides coding examples in the form of complete and annotated programs.

The appendices contain reference information:

- Appendix A summarizes command line syntax, including command switches and their default settings.
- Appendix B summarizes formatting rules suggested for your use.
- Appendix C provides a model template. Appendix D lists current implementation limits.
- Appendix E contains error messages generated by the compiler.
- Appendix F illustrates a sample output listing.
- Appendix G describes methods for interfacing with BLISS-10.
- Appendix H describes the use of user-generated OTS files.

RELATED MANUALS

BLISS Language Guide (AA-H275C-TK)

This manual completely describes the BLISS-16, BLISS-32, and BLISS-36 languages. It can be used both as a learning tool for the languages and as a reference guide.

BLISS Language Guide Update Package (AD-H275C-T1)

The update package provides Version 4.0 of the BLISS Language Guide.

BLISS Primer (order from Educational Services)

The BLISS Primer is a guide to a self-paced BLISS learning course. The language features are described and exemplified. Each section is followed by a quiz and suggested solutions. This document is strongly recommended for the BLISS novice.

BLISS Pocket Guide (AV-AT45A-TK)

This guide presents a concise syntax summary for the family of BLISS languages. A summary of the command line syntax for the respective compilers is also provided.

XPORT Programmer's Guide (AA-J201A-TK)

The guide is a tutorial and reference manual for the XPORT transportable-programming tools package. XPORT is a collection of source-level tools that provide input/output and operating-system services for BLISS-32 and BLISS-36.

SYNTAX NOTATION

Syntax notation used for defining BLISS-36 is explained thoroughly in Chapter 2 of the BLISS Language Guide. The following is a summary of the syntax notation 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 replicated zero or more times.
item ,...	The item directly preceding the ",..." can be replicated zero or more times, with the items separated by commas.
item+...	The item directly preceding the "+..." can be replicated zero or more times, with the items separated by plus signs.

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

SUMMARY OF TECHNICAL CHANGES

This manual provides BLISS-36 user information for Version 4.0 of the BLISS-36 compiler. This section summarizes technical changes, additions, and deletions to the guide since Version 3.0.

- /CHECK, /CROSS-REFERENCE (DECsystem-20), and /CREF (DECsystem-10) switches have been added as general-qualifiers to the BLISS-36 command lines.
- /MASTER-CROSS-REFERENCE has been added as an output-qualifier to the BLISS-36 compiler for both TOPS-10 and TOPS-20.
- An extended addressing capability has been added to the BLISS-36 compiler for TOPS-20 along with an extended addressing SECTION-INDEPENDENT option.
- A new cross-reference capability (BCREF) has been added to replace BLSCRF.
- Additions have been made to the Programming Considerations chapter (6), which include the use of BUILTIN PC, indexed loop coding errors, the advanced use of BLISS macros, and extended addressing coding differences.
- Changes and additions have been made to the compiler listing formats to update the examples and provide an example of cross-reference listings.
- An Examples chapter (10) has been included to provide additional coding examples.

CHAPTER 1

TOPS-20 OPERATING PROCEDURES

This chapter discusses the TOPS-20 operating procedures used to compile a BLISS program. Compilation, including command-line syntax and semantics, is considered first. Next, file specifications for input to a BLISS-36 compilation are described and illustrated. Finally, the command-line switches relevant to a BLISS-36 compilation are given.

Compiling, linking, and executing a BLISS-36 program is a straightforward procedure. In the simplest case, to compile and execute a program that consists of a single module, you enter the module in a file (for example, ALPHA.B36), compile it with the BLISS-36 compiler, link it using LINK, and then execute the linked image. The EXECUTE command automatically invokes LINK as follows:

```
@BLISS
BLISS>ALPHA
BLISS>/EXIT
@EXECUTE ALPHA
```

The first command invokes the BLISS compiler to compile the module in the file ALPHA.B36 and to produce an object file ALPHA.REL. The second command uses the object module in the file ALPHA.REL to produce an executable image in memory and to execute the image.

To save the linked image, invoke LINK explicitly and save the resulting image as follows:

```
@BLISS
BLISS>ALPHA/EXIT
@LOAD ALPHA
@SAVE ALPHA
```

You can control the compiler by using command-line switches. These switches add a level of complexity to the compilation process, but they also provide a significant number of options by which you can vary the performance of the compiler in the production of output, the formatting of listings, and the degree of optimization to be performed.

1.1 COMPILING A BLISS PROGRAM

The BLISS compiler uses the standard TOPS-20 command interpreter, the COMND% JSYS, to parse the command line. As such, various features of command line processing that are common to many programs and EXEC commands on TOPS-20 are also common to the BLISS compiler. Some of these include command recognition, file specification completion, editing characters, and the question mark character (?). Refer to the TOPS-20 User's Guide for a description of these facilities.

To compile a BLISS program, you run the BLISS compiler from the command level and wait for the 'BLISS>' prompt. (The simplest way to run the BLISS compiler is to have the compiler, BLISS.EXE, reside on logical device SYS:; for the rest of this chapter, the compiler is assumed to be invoked by typing 'BLISS' at the command level.) Input specs and global switches, if supplied, are then supplied.

Global switches apply to all input specs and precede them in the command line. They override default switch settings. Input-specs consist of one or more file names followed by switch settings that apply to an individual file or concatenated file. Switch settings in an input-spec override global switch settings.

- To compile a program, use the following command:

  ```
  BLISS>MYPROG
  ```

  The BLISS compiler uses the file MYPROG.B36 or MYPROG.BLI as its input, compiles the source in that file, and produces object file MYPROG.REL.

- To produce a listing file, use the output switch /LISTING:

  ```
  BLISS>MYPROG/LISTING
  ```

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

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

  ```
  BLISS>ALPHA,BETA,GAMMA
  ```

  The compiler compiles ALPHA.B36, producing the object file ALPHA.REL; then BETA.B36, producing BETA.REL; and then GAMMA.B36, producing GAMMA.REL.

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

  ```
  BLISS>ALPHA+BETA+GAMMA
  ```

  The BLISS compiler compiles the program formed by the concatenation of ALPHA.B36, BETA.B36, and GAMMA.B36, and produces the single object file ALPHA.REL.

- To perform a multifile compilation in which one command-line switch is common to all source files in an input-spec, include the appropriate switch before the input-spec:

  ```
  BLISS>/LIBRARY ALPHA,BETA,GAMMA/NOLIBRARY,DELTA
  ```

  The command line consists of four input-specs, which must be separated by commas. Placing the /LIBRARY switch before the first input-spec has the effect of overriding the default switch settings of /NOLIBRARY and /OBJECT for all input-specs, unless it, in turn, is superseded by a switch setting that applies to an individual input-spec. A switch setting that follows an input-spec applies only to that input-spec. Thus, the command line above causes the compiler to produce three library files and one object file: ALPHA.L36, BETA.L36, DELTA.L36, and GAMMA.REL.

- To produce an object file with a name different from that of the source file, specify the new object file name in the command

  ```
  BLISS>ALPHA/OBJECT:GAMMA
  ```

  The BLISS compiler produces the object file GAMMA.REL.

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

  ```
  BLISS>ALPHA/LIBRARY
  ```

  The BLISS compiler compiles the input file ALPHA.R36 and produces the library file ALPHA.L36.

NOTE

The TOPS-20 EXEC does not support BLISS-36 in LOAD-class commands. Therefore, the following commands will not compile ALPHA as a BLISS-36 module. However, they will attempt to use BLISS-10 to compile ALPHA.BLI.

```
@EXECUTE ALPHA.BLI
@LOAD ALPHA.BLI
```

1.1.1 Command-Line Syntax

bliss-compilation	BLISS>bliss-command-line ...
bliss-command-line	{ switch ... } space input-spec ,...
input-spec	file-spec+... { switch ... }
space	blank ...
switch	{ output-switch general-switch check-switch terminal-switch optimization-switch listing-switch reference-switch environment-switch }

1.1.2 Command-Line Semantics

The BLISS-36 compiler uses switches given in the bliss-command-line to modify the initial defaults for each compilation. Then, the concatenated input is compiled in the context of the initial defaults. The switches and the initial default for each switch are described in Section 1.3.

Unless a switch to change the compiler's behavior is given, the output from a compilation initiated from your terminal or batch file is the object file; no listing is generated.

The compiler begins processing with the first file given 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 switches 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+ETA+THETA,OMEGA
```

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

```
BLISS>/LIBRARY ALPHA,BETA/LIST,IOTA
```

If no command-line switches exist in a command line, default switch settings are assumed for all input-specs in the command line. All switches 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 switches.

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. More precisely, file specifications can occur in four contexts:

- In the input-specs of a bliss-command-line
- As the values of the switches /OBJECT, /LIBRARY, /LISTING, or /MASTER-CROSS-REFERENCE
- In REQUIRE or LIBRARY declarations in the module being compiled
- In the object-time system (OTS) module switch (for TOPS-10 style file specs only)

The file-spec is a standard TOPS-20 file specification, as described in the DECSYSTEM-20 User's Guide, and is interpreted as follows:

1. Logical name translation occurs.
2. If a file type is not given, a default file type is used (see below).
3. If the file-spec applies to an output file and a file name is not given, the name of the first input file in the input-spec is used.

This same interpretation is also used by the compiler when it processes the file specification given in a REQUIRE or LIBRARY declaration.

The file-spec must be fully specified in the OTS module switch. That is, no defaults are applied by the compiler.

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 list the compiler applies depends on the output specified for the compilation, as indicated in the following list:

Input-Spec Used to Produce	Default Type List
An object module	.B36, .BLI
A library file	.R36, .REQ, .B36, .BLI

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

File Use	Default Type List
File given in a REQUIRE declaration	.R36, .REQ, .B36, .BLI
File given in a LIBRARY declaration	.L36

For example, suppose you have entered the following program 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
```

Since the bliss-command-line contains no switch 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.B36, then, not finding that file, 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. Since 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.R36, then CBLISS.REQ, then CBLISS.B36, and finally CBLISS.BLI. When the compiler processes the LIBRARY declaration, it uses the default type list associated with library declarations and searches for TBLISS.L36.

1.3 COMMAND-LINE SWITCHES

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

- Output switch -- defines the types of output to be produced
- General switch -- sets a %VARIANT value and specifies code and debug information

- Check switch -- controls the level of semantic checking done during compilation.
- Terminal switch -- controls output produced on a terminal
- Optimization switch -- supplies code optimization strategies and directions
- Listing switch -- provides output listing information concerning the source and machine code
- Reference switch -- includes cross-reference information in output listing and/or a master cross-reference data file
- Environment switch -- identifies the processor model and target operating system of the generated code

1.3.1 Output Switches

Output switches are used to indicate the type of output to be produced from a BLISS-36 compilation and to give names for the files to be produced when you do not want to use the default names. Some examples of output switches are given in the following list:

- To suppress the production of an object file, use the /NOOBJECT switch in the bliss-compilation, as follows:

  ```
  BLISS>ALPHA/NOOBJECT
  ```

  The BLISS-36 compiler reads the source in the file ALPHA.B36 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 /LISTING switch, as follows:

  ```
  BLISS>ALPHA/LISTING
  ```

  The BLISS-36 compiler produces an object file ALPHA.REL and a list file ALPHA.LST.

  However, to obtain list files in a multifile compilation, use the /LISTING switch before the input-specs, as follows:

  ```
  BLISS>/LISTING ZETA,ETA,THETA
  ```

  The BLISS-36 compiler generates both object and list files for each input file.

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

  ```
  BLISS>ALPHA/OBJECT:BETA/LISTING:GAMMA
  ```

  The compiler reads the input file ALPHA.B36, and produces the object file BETA.REL and the list file GAMMA.LST.

- To produce a master cross-reference data file, use the following:

  ```
  BLISS>ALPHA/MASTER-CROSS-REFERENCE:MASTER
  ```

The compiler reads the input file ALPHA.B36, and produces the object file ALPHA.REL and master cross-reference file MASTER.CRF.

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

```
BLISS>ALPHA/LIBRARY
```

The compiler reads the input file ALPHA.B36 and produces the library file ALPHA.L36.

1.3.1.1 Syntax - Output-switch syntax is:

```
output-switch  { /OBJECT {:file-spec}                    | /NOOBJECT                    }
               { /LISTING {:file-spec}                   | /NOLISTING                   }
               { /LIBRARY {:file-spec}                   | /NOLIBRARY                   }
               { /MASTER-CROSS-REFERENCE {:file-spec}    | /NOMASTER-CROSS-REFERENCE    }
```

The compiler can produce either a library or an object file, but not both. Therefore, the switches /OBJECT and /LIBRARY must not be applied to the same input-spec.

1.3.1.2 Defaults - In the absence of an explicit choice of output switch, the following are assumed:

```
/OBJECT   /NOLISTING   /NOLIBRARY   /NOMASTER-CROSS-REFERENCE
```

If a file-spec is not given, the file name of the first file-spec in the input-spec is combined with the default file type to form the file-spec. If a file-spec is given but the file-spec does not include a file type, the following defaults are supplied, according to the file-designator:

File-Designator	Default Type
/OBJECT	.REL
/LISTING	.LST
/LIBRARY	.L36
/MASTER-CROSS-REFERENCE	.CRF

1.3.1.3 Semantics - Output switches have the following interpretation:

/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.REL'.
/NOOBJECT	Do not produce an object file.
/LISTING:file-spec	Produce a list file in the file specified by file-spec.

/LISTING	Produce a list file in the file specified by 'input-file-name.LST'.
/NOLISTING	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.L36'.
/NOLIBRARY	Do not produce a library file.
/MASTER-CROSS-REFERENCE:file-spec	Produce a master cross-reference file in the file specified by file-spec.
/MASTER-CROSS-REFERENCE	Produce a master cross-reference file in the file specified by 'input-file-name.CRF'.
/NOMASTER-CROSS-REFERENCE	Do not produce a master cross-reference file.

1.3.2 General Switches

General switches are used to specify code and debug information and to set the value for the lexical function %VARIANT. Some examples of using general switches follow:

- To include the necessary debug linkage in the compiled program, use the /DEBUG switch in the bliss-compilation:

  ```
  BLISS>ALPHA/DEBUG
  ```

  The compiler reads the source from ALPHA.BLI and creates an object file ALPHA.REL, which includes additional code for interface with SIX12.

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

  ```
  BLISS>ALPHA/NOCODE
  ```

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

  ```
  BLISS>ALPHA/VARIANT:17
  ```

- To limit the number of errors diagnosed to 10, use the /ERROR-LIMIT switch as follows:

  ```
  BLISS>ALPHA/ERROR-LIMIT:10
  ```

1.3.2.1 Syntax - General-switch syntax is:

```
                    /DEBUG      |  /NODEBUG
                    /CODE       |  /NOCODE
general-switch      /VARIANT { : value }
                    /ERROR-LIMIT { : value }
                    /EXIT
```

1.3.2.2 Defaults - In the absence of explicit choices of general switches, the following are assumed:

/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 0.

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

If the /ERROR-LIMIT is given without a specified value, a value of 1 is assumed.

1.3.2.3 Semantics - The interpretation of the command switches is given in the following list:

/DEBUG — Generate debugging linkage and do not do certain optimization so that a user may effectively use SIX12. Also, include symbolic information in the object file produced, and maintain the frame pointer (FP) in routine prologs and epilogs.

/NODEBUG — Produce symbolic information but no debug linkage, and do not limit optimizations for effective use of SIX12.

/CODE — Generate object code for the BLISS source module.

/NOCODE — Perform only a syntax check of the program.

/VARIANT — Set %VARIANT to 1.

/VARIANT:n — Set %VARIANT to n, where n is a decimal integer in the range:

$$-(2^{**}35) \leq n \leq (2^{**}35)-1$$

/EXIT — Terminate the compiler operation and returns control to the system. /EXIT can appear in a command line or on a separate line.

/ERROR-LIMIT — Set error limit to 1.

/ERROR-LIMIT:n — Limit to n the number of errors diagnosed before terminating compilation.

1.3.3 Check Switch

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

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

```
BLISS>ALPHA/CHECK:NOFIELD
```

- To check for the use of uninitialized storage, use the check switch as follows:

```
BLISS>ALPHA/CHECK:INITIAL
```

1.3.3.1 Syntax - Check switch syntax is defined as follows:

check switch	/CHECK: { (check-value ,...) / check-value }
check-value	{ FIELD / INITIAL / OPTIMIZE / REDECLARE } | { NOFIELD / NOINITIAL / NOOPTIMIZE / NOREDECLARE }

1.3.3.2 Defaults - In the absence of specific choices of check-values, the following values are assumed by default:

```
FIELD    INITIAL    OPTIMIZE    NOREDECLARE
```

1.3.3.3 Semantics - The /CHECK switch 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 storage.
NOINITIAL	Do not check for 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 %X'FF'`

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.

1.3.4 Terminal Switches

Terminal switches are 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 using the terminal switches are as follows:

- To see the statistics for each routine as they are produced during the compilation, use the /STATISTICS switch, as follows:

  ```
  BLISS>ALPHA/STATISTICS
  ```

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

  ```
  BLISS>ALPHA/STATISTICS/NOERRS
  ```

  /NOERRS is useful when there are too many errors to be listed on the terminal and the user is requesting a listing.

1.3.4.1 Syntax - Terminal-switch syntax is:

terminal-switch	{ /ERRS | /NOERRS } { /STATISTICS | /NOSTATISTICS }

1.3.4.2 Defaults - In the absence of explicit choices of terminal switches, the following are assumed:

```
/ERRS  /NOSTATISTICS
```

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

1.3.4.3 Semantics - The terminal switches have the following meanings:

Switch	Meaning
/ERRS	List each error on the terminal as it is encountered in the compilation.
/NOERRS	Do not list errors on the terminal.
/STATISTICS	List the name and size (in words) of each routine on the terminal after each routine is compiled.
/NOSTATISTICS	Do not list routine names and sizes.

1.3.5 Optimization Switches

Optimization switches are 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 using the optimization switches are as follows:

- To increase the compilation speed by omitting some standard optimizations, use the /QUICK switch, as follows:

```
BLISS>ALPHA/QUICK
```

- To get minimum optimization, use the /OPTLEVEL switch with the value 0, as follows:

```
BLISS>ALPHA/OPTLEVEL:0
```

- To obtain maximum optimization, use the /OPTLEVEL switch with the value 3, as follows:

```
BLISS>ALPHA/OPTLEVEL:3
```

- To direct the compiler to use techniques that may use more storage for the program to increase its operating speed, use the /ZIP switch, as follows:

```
BLISS>ALPHA/ZIP
```

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

```
BLISS>ALPHA/NOSAFE
```

A detailed discussion of the optimizations resulting from using the optimization switches is given in Chapter 8.

1.3.5.1 Syntax - Optimization-switch syntax is:

optimization-switch	(optimize-switch | optlevel-switch | safe-switch | zip-switch | quick-switch)
optimize-switch	{ /OPTIMIZE | /NOOPTIMIZE }
optlevel-switch	/OPTLEVEL : { 0 | 1 | 2 | 3 }
safe-switch	{ /SAFE | /NOSAFE }
zip-switch	{ /ZIP | /NOZIP }
quick-switch	{ /QUICK | /NOQUICK }

1.3.5.2 Defaults - In the absence of an explicit optimization switch, the following are assumed:

/NOQUICK /NOZIP /OPTLEVEL:2 /SAFE /OPTIMIZE

The compiler is directed:

- To perform normal optimization, balancing the time/space trade-off in favor of space
- To assume that all variables are addressed by name
- To perform optimization across mark points
- To perform flow analysis. (Refer to Section 8.1.2.2.)

1.3.5.3 Semantics - Optimization switches indicate that one or more optimize options are specified. The optimize switches have the following meanings.

Optimize-Value	Meaning
/QUICK	Omit some standard optimizations to increase the compilation speed.
/NOQUICK	Include standard optimizations.
/ZIP	Increase the execution efficiency of the program being compiled by using more space where appropriate. For more information on the effect of this value, see Section 8.1.4.
/NOZIP	Do not increase the space occupied by the program to improve its operating speed. For more information on the effect of this value, see Section 8.1.2.2.
/OPTLEVEL:n	Optimize the program being compiled according to the optimize-level n, as follows:

Optimize-Level	Meaning
0	Minimum optimization
1	Subnormal optimization
2	Normal optimization
3	Maximum optimization

n=3 optimizes speed at the expense of space in the same way as /ZIP. /OPTLEVEL:3 is equivalent to /OPTLEVEL:2/ZIP. For more information on the effect of this value, see Section 8.1.2.

/SAFE	Assume that all named data-segments are referenced by name and not manipulated in any way indirectly, and use optimization techniques that exploit this fact. For more information on the effect of this value, see Section 8.1.2.1.
/NOSAFE	Assume that sometimes a named data-segment is referenced by means of a computed expression and, therefore, some optimization techniques cannot be used.
/OPTIMIZE	Perform full flow analysis over an entire routine.
/NOOPTIMIZE	Restrict flow analysis so that all data is assumed to be changed across mark points. (See Chapter 8 for a more complete discussion.)

1.3.6 Listing Switches

Listing switches are used to supply information about the form of the output listing and are used in conjunction with the /LISTING output switch. Some examples of using the listing switches follow:

- To obtain a paged listing with 44 lines on each page, give the following command line:

  ```
  BLISS>ALPHA/LISTING/PAGSIZ:44
  ```

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

  ```
  BLISS>ALPHA/LISTING/NOHEADER/FORMAT:EXPAND
  ```

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

  ```
  BLISS>ALPHA/LISTING/FORMAT:REQUIRE
  ```

- To obtain an output listing that is intended to be assembled by the MACRO assembler, use the ASSEMBLY option, as follows:

  ```
  BLISS>ALPHA/LISTING/FORMAT:ASSEMBLY
  ```

- To obtain a listing that is intended to be assembled and that does not contain binary, include the NOBINARY option:

  ```
  BLISS>ALPHA/LISTING/FORMAT:(ASSEMBLY,NOBINARY)
  ```

The form of the output listing is described in Section 3.2.

1.3.6.1 Syntax - Listing-switch syntax is:

listing-switch	/PAGSIZ: number-of-lines /HEADER | /NOHEADER /UNAMES | /NOUNAMES /FORMAT : format-option-list
number-of-lines	{ 20 | 21 | 22 | ... | 52 }
format-option-list	(option, ...) option
option	ASSEMBLY | NOASSEMBLY BINARY | NOBINARY COMMENTARY | NOCOMMENTARY EXPAND | NOEXPAND LIBRARY | NOLIBRARY OBJECT | NOOBJECT REQUIRE | NOREQUIRE SOURCE | NOSOURCE SYMBOLIC | NOSYMBOLIC TRACE | NOTRACE

1.3.6.2 Defaults - In the absence of an explicit choice of listing switches, the following are assumed:

```
/PAGSIZ:52    /NOUNAMES    /NOHEADER
/FORMAT:(NOASSEMBLY,BINARY,COMMENTARY,NOEXPAND,NOLIBRARY,
         OBJECT,NOREQUIRE,SOURCE,SYMBOLIC,NOTRACE)
```

The compiler produces a listing with 52 lines on each page; the listing includes no expansion or tracing. The listing resembles a typical macro source file but cannot be assembled.

1.3.6.3 Semantics - The listing switches indicate that one or more listing options are given for the compilation. Command-line switches are preceded by a slash, while switches that can appear as a part of the /FORMAT switch are not. 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.
/PAGSIZ:lines	Use the number of lines specified for each page of the list file. The number of lines must lie in the range: 20 < lines < 52.
/UNAMES	Replace names by machine-generated names so that all names are unique and independent of scope, resulting in a listing that can be correctly assembled.
/NOUNAMES	Do not replace names by unique names.
/FORMAT:	One (or more in parentheses) of the following options:

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 3.2.4.2.

NOLIBRARY

Do not produce a trace identifying any libraries and their contributions.

REQUIRE

Include the contents of the specified file in the listing file. For an example, see Section 3.2.4.2.

NOREQUIRE

Do not include the contents of the specified REQUIRE file in the listing.

EXPAND

Include the expansion of each macro call in the listing file. For an example of a macro expansion, see Section 3.2.4.3.

NOEXPAND

Do not include the expansion of macros.

TRACE

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 3.2.4.4.

NOTRACE

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.

OBJECT

Produce the object part of the output listing.

NOOBJECT

Suppress the object part of the output listing.

ASSEMBLY

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.

NOASSEMBLY

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.

Each of the code-values is described and illustrated in Section 3.2.2 in connection with the discussion of the output listing produced by a BLISS compilation. Understanding the purpose of these code-values requires knowledge of the format and purpose of the output listing, as discussed in that section.

1.3.7 Reference Switches

The reference switches allow a cross-reference listing to be included with the compiler listing, and/or a cross-reference data file to be created (refer to Section 1.3.1) to produce a master cross-reference listing (refer to master cross-reference utility program BCREF in Section 9.3). Some examples of using the reference switches are as follows:

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

  ```
  BLISS>ALPHA/LISTING/CROSS-REFERENCE
  ```

  The compiler produces an object file and list file ALPHA.LST, 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>ALPHA/LIST/FORMAT:(NOSOURCE,NOOBJECT)/CROSS-REF
  ```

  The compiler produces an object file and list file ALPHA.LST, which contains only the cross-reference listing.

- To create only a master cross-reference data file, use the /MASTER-CROSS-REFERENCE switch as follows:

  ```
  BLISS>ALPHA/MASTER-CROSS-REFERENCE
  ```

  The compiler produces an object file and, as suitable input for BCREF, master cross-reference file ALPHA.CRF.

- To produce a compiler listing which includes a cross-reference listing, and a master cross-reference data file, use the following:

  ```
  BLISS>ALPHA/LIST/CROSS-REFERENCE/MASTER-CROSS-REFERENCE
  ```

  The compiler produces an object file and list file ALPHA.LST, to which a cross-reference listing is appended, and master cross-reference file ALPHA.CRF.

- 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>ALPHA/LIST/CROSS-REFERENCE:(MULTIPLE)
```

  The compiler produces an object file and list file ALPHA.LST, to which a cross-reference listing is appended that includes multiple references to the same symbol.

1.3.7.1 Syntax - Reference switch syntax is defined as follows:

```
reference-switch:
    /CROSS-REFERENCE [:{ (reference-value ,...) | reference-value }]
    /MASTER-CROSS-REFERENCE [:file-spec]

reference-value:
    { MULTIPLE | NOMULTIPLE }
```

1.3.7.2 Defaults - In the absence of an explicit choice of reference value, the following value is assumed by default:

```
NOMULTIPLE
```

1.3.7.3 Semantics - The /CROSS-REFERENCE switch indicates that a reference-value may be given for the compilation. The reference-value has the following meaning:

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

1.3.8 Environment Switches

Environment switches are used to specify the processor model and the operating system of the target system for which code is to be generated. Some examples of using the environment switches are as follows:

- To generate code that uses instructions available only on a KL10 processor, use the following command line:

```
BLISS>ALPHA/KL10
```

The compiler reads the source from ALPHA.B36 or ALPHA.BLI and creates an object file ALPHA.REL, which makes use of KL10 instructions, such as ADJSP, to make stack adjustments and the EXTEND instruction to implement various character-handling functions.

- To generate code that makes calls on the TOPS-20 monitor using JSYS instructions, use the following command-line:

  BLISS>ALPHA/TOPS20

  If ALPHA contains the main routine of the program, the compiler generates a RESET% JSYS before the call to the main routines and a HALTF% JSYS immediately after the call to the main routine.

1.3.8.1 Syntax - Environment-switch syntax is:

```
environment-switch    { /KA10 | /KI10 | /KL10 | /KS10 }
                      { /TOPS10     | /TOPS20        }
                      { /EXTENDED   | /NOEXTENDED    }
                      { /EXTENDED : SECTION-INDEPENDENT }
```

1.3.8.2 Defaults - If no environment switches are specified, the following are assumed:

/KL10 /TOPS20 /NOEXTENDED

1.3.8.3 Semantics - Environment switches identify the target system for which code is being generated. Environment switches have the following meanings:

/KA10	Generate code that uses only the KA10 instruction set; this code executes on all processor models.
/KI10	Generate code that uses only the KI10 instruction set; this code executes on KI10, KL10, and KS10 processor models.
/KL10	Generate code that uses the KL10 instruction set; this code executes on KL10 and KS10 processor models.
/KS10	Generate code that uses the KS10 instruction set; this code executes on the KI10 and KS10 processor models.
/TOPS10	Generate code that makes calls to the TOPS-10 monitor.
/TOPS20	Generate code that makes calls to the TOPS-20 monitor.
/EXTENDED	Give partial support for the extended addressing option for the KL10 Model B. This switch is valid only when the /KL10 and /TOPS20 switches are specified or implied.

/EXTENDED:SECTION-INDEPENDENT — The same as /EXTENDED, except code is generated that can be executed from any section.

NOTE

A compiler-state-function is a lexical-function that expands to a numeric-literal of 1 or 0 during compilation to indicate whether a certain condition exists. The %SWITCHES lexical-function can be tested during compilation to determine the setting of one or more environment switches. For example, the following command line causes the %SWITCHES function to return the indicated numeric-literal:

```
BLISS>ALPHA/KL10/TOPS10


%SWITCHES(KA10)   - 0
%SWITCHES(KI10)   - 0
%SWITCHES(KL10)   - 1
%SWITCHES(KS10)   - 1
%SWITCHES(TOPS10) - 1
%SWITCHES(TOPS20) - 0
```

For additional information, refer to "Module-Switches" in the BLISS Language Guide.

The preceding discussion also relates to the options of the same name to the module-head switch ENVIRONMENT.

1.3.9 Placement of Switches

Some directions can be given to the compiler either by command-line switches or by switch settings contained in the module being compiled. In some cases, the command-line switch name is the same as the switch name contained in the module (module switches and SWITCHES declaration), and in other cases, it is similar but not identical. Names of common switches are given in Table 1-1.

1.3.10 Switches and Default Module Switch Settings

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

Suppose you are compiling two programs. The first program ALPHA.BLI has a module switch CODE. The second program BETA has no switches. The bliss-command-line is as follows:

```
BLISS>/NOCODE ALPHA,BETA
```

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

Table 1-1: Command Line, Module Switch, and SWITCH Names on TOPS-20

Command Line Name	Module Switch Name	SWITCHES Name
/CHECK	n/a	n/a
/CODE	CODE	n/a
/CROSS-REFERENCE	n/a	n/a
/DEBUG	DEBUG	n/a
/EXTENDED{:ext-option}[1]	ENVIRONMENT(EXTENDED{:ext-option}[1])	n/a
/ERRS	ERRS	ERRS
/FORMAT:ASSEMBLY	LIST(ASSEMBLY)	LIST(ASSEMBLY)
/FORMAT:BINARY	LIST(BINARY)	LIST(BINARY)
/FORMAT:COMMENTARY	LIST(COMMENTARY)	LIST(COMMENTARY)
/FORMAT:EXPAND	LIST(EXPAND)	LIST(EXPAND)
/FORMAT:LIBRARY	LIST(LIBRARY)	LIST(LIBRARY)
/FORMAT:OBJECT	LIST(OBJECT)	LIST(OBJECT)
/FORMAT:REQUIRE	LIST(REQUIRE)	LIST(REQUIRE)
/FORMAT:SOURCE	LIST(SOURCE)	LIST(SOURCE)
/FORMAT:SYMBOLIC	LIST(SYMBOLIC)	LIST(SYMBOLIC)
/FORMAT:TRACE	LIST(TRACE)	LIST(TRACE)
/KA10	ENVIRONMENT(KA10)	n/a
/KI10	ENVIRONMENT(KI10)	n/a
/KL10	ENVIRONMENT(KL10)	n/a
/KS10	ENVIRONMENT(KS10)	n/a
/MASTER-CROSS-REFERENCE	n/a	n/a
/OPTLEVEL:n	OPTLEVEL=n	n/a
/SAFE	SAFE	SAFE
/TOPS10	ENVIRONMENT(TOPS10)	n/a
/TOPS20	ENVIRONMENT(TOPS20)	n/a
/UNAMES	UNAMES	UNAMES
/ZIP	ZIP	ZIP

1. The EXTENDED{:ext-option} implies EXTENDED:SECTION-INDEPENDENT for the command-line and EXTENDED:SECTION_INDEPENDENT for the module switch.

n/a (not applicable) indicates that no corresponding switch exists.

1.3.11 Positive and Negative Forms of Switches

In general, two forms of a switch 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. Positive and negative forms of a switch are mutually exclusive; only one form for any switch should be given in a bliss-command-line.

1.3.12 Abbreviations of Switch and Value Names

The command switch names and value names can be abbreviated as long as the COMND% JSYS can complete the command unambiguously. This is true both for switches that can take values and switches that take no value. (Refer to Appendix A for a summary of positive and negative forms of switches and values.)

1.4 SPECIAL FEATURES

1.4.1 Indirect Files

An indirect file is a file referenced within a BLISS command line; it is used to complete a BLISS command. The indirect file may contain a complete or partial BLISS command line: for example, file names and switch settings. You reference the file by specifying an "at sign" (@) followed by the file-spec, the contents of which expand and complete the command line. For example, if the file TTY.CMD contains one line with the following switch settings:

```
/LISTING:TTY:/FORMAT:(NOBINARY,NOCOMMENTARY)/NOHEADER
```

and the following command, using the indirect file TTY.CMD to specify the remainder of a command line, is issued:

```
BLISS>ALPHA@TTY.CMD
```

The compiler compiles ALPHA and sends the listing to the terminal without the binary, commentary, and page headers. This is a convenient shorthand method of specifying a commonly used set of switches.

For another example, assume that the file LIB.CMD contains the following command line:

```
LIB1+LIB2+LIB3/LIBRARY:LIB.L36
```

and that a reference to the indirect file is specified in a BLISS command line:

```
BLISS>@LIB.CMD
```

The compiler uses the concatenation of files LIB1, LIB2, and LIB3 as input and produces a library file, LIB.L36.

1.4.2 EXEC Command

A bliss-command-line may be specified on the same line as that of a command invoking the compiler as follows:

```
@BLISS bliss-command-line
```

The compiler will be invoked, compile the specified file or files, and then exit back to the EXEC. Command recognition, file specification completion, and the question mark character features are not available in this mode of operation.

CHAPTER 2

TOPS-10 OPERATING PROCEDURES

This chapter discusses the TOPS-10 operating procedures used to compile a BLISS program. The form of the command line is considered first. Then, the input to a BLISS-36 compilation is described and illustrated. Finally, the command-line switches relevant to a BLISS-36 compilation are given.

Compiling, linking, and executing a BLISS-36 program is a straightforward procedure. In the simplest case, to compile and execute a program that consists of a single module, you enter the module in a file (for example ALPHA.B36) and then compile it with the BLISS-36 compiler, link it using LINK, and then run the linked image. The EXECUTE command automatically invokes LINK as follows:

```
.R BLISS
*ALPHA=ALPHA
*^C
.EXECUTE ALPHA
```

The first command invokes the BLISS compiler to compile the module in the file ALPHA.B36 and to produce an object file ALPHA.REL. The second command uses the object module in the file ALPHA.REL to produce an executable image in memory and to execute the image.

To save the linked image, use the LOAD command and save the resulting image as follows:

```
.R BLISS
*ALPHA=ALPHA
*^C
.LOAD ALPHA
.SAVE ALPHA
```

You can control the compiler by using command-line switches. These switches add a level of complexity to the compilation process, but they also provide a significant number of options by which you can vary the performance of the compiler in the production of output, the formatting of listings, and the degree of optimization performed.

2.1 COMPILING A BLISS PROGRAM

The BLISS compiler uses the standard TOPS-10 command interpreter, SCAN, to parse the command line. As such, various features of command line processing that are common to many programs and the TOPS-10 monitor are also common to the BLISS compiler. Some of these include formats for file specifications and common switches.

To compile a BLISS program, you run the BLISS compiler from the command level and wait for the '*' prompt. (The simplest way to run the BLISS compiler is to have the compiler, BLISS.EXE, reside on logical device SYS:. For the remainder of this chapter, it is assumed that you have invoked the compiler by typing 'R BLISS' in monitor mode.) You then provide a source-file list, an optional output-file-list, and the number of switches desired; some examples are given in the following list:

- To compile a program, give the following command:

  ```
  *MYPROG=MYPROG
  ```

  The BLISS compiler uses the file MYPROG.B36 or MYPROG.BLI as its input, compiles the source in that file, and produces object file MYPROG.REL.

- To produce a listing file, specify the listing file as follows:

  ```
  *MYPROG,MYPROG=MYPROG
  ```

  The BLISS compiler produces, in addition to the object file, listing file MYPROG.LST.

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

  ```
  *GAMMA=ALPHA
  ```

  The BLISS compiler produces the object file GAMMA.REL.

- To produce a library file instead of an object file, use the command switch /LIBRARY as shown in the following:

  ```
  *ALPHA=ALPHA/LIBRARY
  ```

  The BLISS compiler compiles input file ALPHA.R36 and produces library file ALPHA.L36.

- To compile a program that consists of several pieces, each in a separate file, include all file names on the command line

  ```
  *ALPHA=ALPHA,BETA,GAMMA
  ```

  The BLISS compiler compiles the program formed by the concatenation of ALPHA.B36, BETA.B36, and GAMMA.B36, and produces the single object file ALPHA.REL.

NOTE

The TOPS-10 EXEC does not support BLISS-36 in COMPIL-class commands. Therefore, the command

```
.EXECUTE ALPHA.BLI
```

will not compile and execute ALPHA as a BLISS-36 module. However, it will attempt to use BLISS-10 to compile ALPHA.BLI.

2.1.1 Command-Line Syntax

bliss-command-line	{output-file-list} = source-file-list {switch...}
source-file-list	source-file-spec,...
output-file-list	{object-file-spec ,... } {,listing-file-spec ,...} {,master-cref-spec }
source-file-spec object-file-spec listing-file-spec master-cref-spec	file-spec
file-spec	See Section 2.2
switch	See Section 2.4

2.1.2 Command-Line Semantics

The BLISS-36 compiler uses any switches given in the bliss-command-line to modify the initial defaults for each compilation. Then, the concatenated input is compiled in the context of the initial defaults. The switches and the initial default for each switch are described in Section 2.4.

Unless a switch is used to change the compiler's behavior, the output compilation initiated from your terminal (or batch file) will consist of an object file (if specified), a listing file (if specified), and a terminal listing.

The compiler begins with the first file given and continues until an end-of-file is reached. The compiler continues reading input from the next file specified, and so on, until all the files in the source file list are used.

2.2 FILE SPECIFICATIONS

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

- The source-file-list of a bliss-command-line
- REQUIRE and LIBRARY declarations in the module being compiled
- The OTS module switch

2.2.1 Syntax

The standard TOPS-10 file specification is:

file-spec	{ device: } file-name { .extension } { [ppn] }
device	any logical or physical device name of 1 to 6 alphanumeric characters
file-name	1 to 6 alphanumeric characters
extension	0 to 3 alphanumeric characters
ppn	project-number,programmer-number

2.2.2 Semantics

A file specification is interpreted as follows:

1. If an extension is not given a default extension is used, as described in the next section.

2. If the file-spec applies to an output file and a file name is not given, the name of the first input file in the source-file-list is used.

This same interpretation is also used by the compiler when processing the file specification given in a REQUIRE or LIBRARY declaration.

The file-spec must be fully specified in the OTS module switch. That is, no defaults are supplied by the compiler.

2.2.3 Default Extension

The compiler has two ordered lists of default extensions to be tried for a source-file-spec that does not include an extension. The list that the compiler applies depends on the output specified for the compilation, as indicated in the following list:

File Use	Default Extension List
Input-spec used to produce an object module	.B36, .BLI
Input-spec used to produce a library file	.R36, .REQ, .B36, .BLI

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

File Use	Default Extension List
File given in a REQUIRE declaration	.R36, .REQ, .B36, .BLI
File given in a Library declaration	.L36

For example, suppose you have entered the following program in the file ALPHA.BLI:

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

And, suppose further that you compile it as follows:

```
*ALPHA=ALPHA
```

Since the bliss-command-line shown does not contain a switch requesting the production of a library file, the output of the compilation is an object module. The compiler, therefore, chooses the list of default extensions associated with object module output and searches first for ALPHA.B36, then, not finding that file, 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. Since an extension for CBLISS is not given, the compiler uses the list of default extensions for files in a REQUIRE declaration and searches for CBLISS.R36, then CBLISS.REQ, then CBLISS.B36, and finally CBLISS.BLI. When the compiler processes the LIBRARY declaration, it uses the default extensions list associated with library declarations and searches for TBLISS.L36.

2.3 OUTPUT SPECIFICATIONS

An output specification is used to indicate the type of output to be produced from a BLISS-36 compilation and to give names for the files to be produced when you do not want to use the default names. There are two ways of specifying output specifications to the compiler. The first is to specify an object or list file specification in the bliss-command-line. The second is to specify both an object file specification and the /LIBRARY switch, in which case the compiler creates a library file, rather than an object file. Thus, the syntax for the output-specification could appear as follows:

```
output-spec  {library-spec  }
             {file-spec-list}  =  ... {library-switch }
             {nothing       }
```

```
file-spec-list  {object-spec ,... }
                {,list-spec ,...  }
                {,master-cref-spec}
```

Some examples of ways to give output specifications to the compiler are given in the following list:

- To suppress the production of an object file, omit the object file specification as follows:

  ```
  *=ALPHA
  ```

  The BLISS-36 compiler reads the source file ALPHA.B36 but produces no output files. However, error messages and summary information are produced at the terminal.

- To obtain a list file, give a listing file specification:

  ```
  *ALPHA,ALPHA=ALPHA
  ```

  The BLISS-36 compiler produces an object file ALPHA.REL and a list file ALPHA.LST.

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

  ```
  *BETA,GAMMA=ALPHA
  ```

  The compiler reads the source file ALPHA.B36 and produces the object file BETA.REL and the list file GAMMA.LST.

- To produce a master cross-reference data file, use the master cross-reference file specification:

  ```
  *,,MASTER=ALPHA
  ```

  The compiler reads the source file ALPHA.B36 and produces the master cross-reference data file MASTER.CRF.

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

  ```
  *ALPHA=ALPHA/LIBRARY
  ```

  The compiler reads the source file ALPHA.B36 and produces the library file ALPHA.L36.

2.4 COMMAND-LINE SWITCHES

The switches in the command line allow you to give the compiler information about the status of the compilation. A library switch, for example, tells the compiler something about the kind of output you want from the compilation. An optimization switch describes the amount and type of optimization to be performed. A source-list switch indicates the switch of the source part of the output, and so on. The kinds of switches are indicated in the following syntax:

```
switch  { library-switch
        { general-switch
        { check-switch
        { terminal-switch
        { optimization-switch
        { listing-switch
        { reference-switch
        { environment-switch
```

2.4.1 Library Switches

Library switches are used to indicate whether an object spec refers to an object file or to a library file. The object-spec can specify either a program's object file or a library file, but not both. The /LIBRARY switch is used to specify that the object-spec refers to a library file.

2.4.1.1 Syntax - Library-switch syntax is:

```
library-switch  { /LIBRARY  |  /NOLIBRARY }
```

2.4.1.2 Defaults - If no output-specification is specified, the following is assumed:

```
*,,=source-file-spec/NOLIBRARY
```

Both the commas and equals sign are unnecessary if the object-spec, list-spec, and master-cref-spec are omitted. The equals sign is required but the commas are not, if only the object-spec is specified. However, both are required if the list-spec and/or the master-cref-spec is specified.

No object file is produced if the object-spec is omitted. No listing file is produced if the list-spec is omitted. And no master cross-reference data file is produced if the master-cref-spec is omitted. /NOLIBRARY is assumed if the library switch is omitted.

If the file type is omitted from a file-spec, the following file-type defaults are supplied, according to the file-designator:

File-Designator	Default Extension
object-spec/NOLIBRARY	.REL
object-spec/LIBRARY	.L36
list-spec	.LST
master-cref-spec	.CRF

2.4.2 General Switches

General switches are used to specify code and debug information and to set the value for the lexical function %VARIANT. Some examples of using general switches follow:

- To include the necessary debug linkage in the compiled program, use the /DEBUG switch in the bliss-compilation:

  ```
  *ALPHA=ALPHA/DEBUG
  ```

  The compiler reads the source from ALPHA.BLI and creates an object file ALPHA.REL, which includes additional code for interface with SIX12.

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

  ```
  *=ALPHA/NOCODE
  ```

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

  ```
  *ALPHA=ALPHA/VARIANT:17
  ```

- To limit the number of errors diagnosed to 10, use the /ERRLIM switch as follows:

  ```
  *ALPHA=ALPHA/ERRLIM:10
  ```

2.4.2.1 Syntax - General-switch syntax is:

```
general-switch    { /DEBUG   | /NODEBUG }
                  { /CODE    | /NOCODE  }
                  { /VARIANT {:value}   }
                  { /ERRLIM  {:value}   }
```

2.4.2.2 Defaults - If no general switches are specified, the following are assumed:

/NODEBUG /CODE /VARIANT:0 /ERRLIM:30

The compiler produces code, does not include the additional debugging information in the object file, and sets the value of %VARIANT to 0.

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

If the general switch /ERRLIM is given without a specified value, a value of 1 is assumed.

2.4.2.3 Semantics - General switches perform the following functions:

/DEBUG	Generate debugging linkage and limit optimization so that SIX12 optimizations may be effectively used. Also, include symbolic information in the object file produced.
/NODEBUG	Produce symbolic information but no debug linkage, and do not limit optimizations (required for the effective use of SIX12).
/CODE	Generate object code for the BLISS source module.
/NOCODE	Perform only a syntax check of the program.
/VARIANT	Set %VARIANT to 1.
/VARIANT:n	Set %VARIANT to n, where n is a decimal integer in the range: $-(2^{**}35) \leq n \leq (2^{**}35)-1$
/ERRLIM	Set limit to 1.
/ERRLIM:n	Limit to n the number of errors diagnosed before terminating the compilation.

2.4.3 Check Switch

The check switch controls the level of semantic checking done during compilation. The switch allows all legal BLISS syntax to be examined for semantic irregularites. Some examples of the use of the check switch 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:

```
*ALPHA=ALPHA/CHECK:NOFIELD
```

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

```
*ALPHA=ALPHA/CHECK:INITIAL
```

2.4.3.1 Syntax - Check switch syntax is defined as follows:

```
check switch      /CHECK:  { (check-value ,... ) }
                           { check-value         }

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

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

```
FIELD    INITIAL    OPTIMIZE    NOREDECLARE
```

2.4.3.3 Semantics - The /CHECK switch 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 storage.
NOINITIAL	Do not check for 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 %X'FF'`
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.

2.4.4 Terminal Switches

Terminal switches are used to control the output that is sent to the terminal. You can have errors or statistics printed or not printed on

the terminal during the compilation of a BLISS program. Some examples of using terminal switches follow:

- To see the statistics for each routine as they are produced during the compilation, use the /STATISTICS switch, as follows:

  ```
  *ALPHA=ALPHA/STATISTICS
  ```

- To suppress error messages, use the /NOERRS switch. As an example, consider the following:

  ```
  *ALPHA=ALPHA/ST/NOERRS
  ```

  Note that the /STATISTICS switch is abbreviated to /ST in the above example.

  /NOERRS is useful in preventing a profusion of error messages from being listed on the terminal when a listing is requested.

2.4.4.1 Syntax - Terminal-switch syntax is:

terminal-switch	{/ERRS | /NOERRS} {/STATISTICS | /NOSTATISTICS}

2.4.4.2 Defaults - If no terminal switches are specified, the following are assumed:

```
/ERRS  /NOSTATISTICS
```

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

2.4.4.3 Semantics - Terminal switches perform the following functions:

Switch	Meaning
/ERRS	List each error on the terminal as it is encountered in the compilation.
/NOERRS	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.

2.4.5 Optimization Switches

Optimization switches are 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 using optimization switches are as follows:

- To increase the compilation speed by omitting some standard optimizations, use the /QUICK switch in the command line, as follows:

  ```
  *ALPHA=ALPHA/QUICK
  ```

- To get minimum optimization, use the /OPTLEVEL switch with the value 0, as follows:

  ```
  *ALPHA=ALPHA/OPTLEVEL:0
  ```

- To obtain maximum optimization, use the /OPTLEVEL switch with the value 3, as follows:

  ```
  *ALPHA=ALPHA/OPTLEVEL:3
  ```

- To direct the compiler to use techniques that may use more storage for the program to increase its operating speed, give the /ZIP switch, as follows:

  ```
  *ALPHA=ALPHA/ZIP
  ```

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

  ```
  *ALPHA=ALPHA/NOSAFE
  ```

A detailed discussion of the optimizations resulting from the use of the optimization switches is given in Chapter 8.

2.4.5.1 Syntax - Optimization-switch syntax is:

optimization-switch	{ optlevel-switch safe-switch zip-switch quick-switch }
optlevel-switch	/OPTLEVEL : optimization-level
optimization-level	{ 0 | 1 | 2 | 3 }
safe-switch	{ /SAFE | /NOSAFE }
zip-switch	{ /ZIP | /NOZIP }
quick-switch	{ /QUICK | /NOQUICK }

2.4.5.2 Defaults - If no optimization switches are specified, the following are assumed:

```
/NOQUICK    /NOZIP    /OPTLEVEL:2    /SAFE
```

The compiler is directed to perform normal optimization, balancing the time/space trade-off in favor of space, to assume that all variables are addressed by name, to perform optimization across mark points, and to perform flow analysis. (See Section 8.1.2.)

2.4.5.3 Semantics - The optimization switches indicate that one or more optimize options are specified. The optimize switches have the following meanings:

Optimize-Value	Meaning
/QUICK	Omit some standard optimizations to increase the compilation speed.
/NOQUICK	Include standard optimizations.
/ZIP	Increase the execution efficiency of the program being compiled by using more space where appropriate. For more information on the effect of this value, see Section 8.1.4.
/NOZIP	Do not increase the space occupied by the program to improve its operating speed. For more information on the effect of this value, see Section 8.1.2.2.
/OPTLEVEL:n	Optimize the program being compiled according to the optimize-level n, as follows:

Optimize-Level	Meaning
0	Minimum optimization
1	Subnormal optimization
2	Normal optimization
3	Maximum optimization

n=3 optimizes speed at the expense of space in the same way as /ZIP. For more information on the effect of this value, see Section 8.1.2.

/SAFE	Assume that all named data-segments are referenced by name and not manipulated in any way indirectly, and use optimization techniques that exploit this fact. For more information on the effect of this value, see Section 8.1.2.1.
/NOSAFE	Assume that sometimes a named data-segment is referenced by means of a computed expression and, therefore, some optimization techniques cannot be used.

2.4.6 Listing Switches

Listing switches are used to supply information about the form of the source code on the output listing. Some examples of using the listing switches are as follows:

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

```
*,ALPHA=ALPHA/PAGSIZ:44
```

- To obtain an unpaged listing in which the macro expansions are given but header information is not, use the following switches:

```
*,ALPHA=ALPHA/LIST:EXPAND/NOHEADER
```

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

```
*,ALPHA=ALPHA/LIST:REQUIRE
```

- To obtain an output listing that is intended to be assembled by the MACRO assembler, use the ASSEMBLY option, as follows:

```
*,ALPHA=ALPHA/LIST:ASSEMBLY
```

- To obtain a listing that is intended to be assembled and that does not contain binary, include the NOBINARY option:

```
*,ALPHA=ALPHA/LIST:(ASSEMBLY,NOBINARY)
```

The form of the output listing is described in Section 3.2.

2.4.6.1 Syntax - Listing-switch syntax is:

```
listing-switch          {/PAGSIZ: number-of-lines }
                        {/HEADER  |  /NOHEADER   }
                        {/UNAMES  |  /NOUNAMES   }
                        {/LIST    : format-option-list}

number-of-lines         {  20  |  21  |  22  | ... |  52  }

format-option-list      {( option, ... )}
                        {option         }

option                  {ASSEMBLY   |  NOASSEMBLY  }
                        {BINARY     |  NOBINARY    }
                        {COMMENTARY |  NOCOMMENTARY}
                        {EXPAND     |  NOEXPAND    }
                        {LIBRARY    |  NOLIBRARY   }
                        {OBJECT     |  NOOBJECT    }
                        {REQUIRE    |  NOREQUIRE   }
                        {SOURCE     |  NOSOURCE    }
                        {SYMBOLIC   |  NOSYMBOLIC  }
                        {TRACE      |  NOTRACE     }
```

2.4.6.2 Defaults - If no listing switches are specified, the following are assumed:

```
/PAGSIZ:52   /NOUNAMES   /NOHEADER
/LIST:(NOASSEMBLY,BINARY,COMMENTARY,NOEXPAND,NOLIBRARY,
       OBJECT,NOREQUIRE,SOURCE,SYMBOLIC,NOTRACE)
```

The compiler produces a listing, with 52 lines on each page, in which no expansion or tracing is included. The listing resembles a typical macro source file.

2.4.6.3 Semantics - Listing switches indicate that one or more listing options 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.

Source-Value	Meaning
/NOHEADER	Do not page the listing, do not include headings, and do not produce statistics in the compilation summary.
/PAGSIZ:lines	Use the number of lines specified for each page of the list file. The number of lines must lie in the range: 20 < lines < 52.
/UNAMES	Replace names by machine-generated names so that all names are unique and independent of scope; the resulting listing can thus be correctly assembled.
/NOUNAMES	Do not replace names by unique names.
/LIST:	One (or more in parentheses) of the following options:

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 3.2.4.2.

NOLIBRARY

Do not produce a trace identifying any libraries and their contributions.

REQUIRE

Include the contents of the specified file in the listing file. For an example, see Section 3.2.4.2.

NOREQUIRE

Exclude the REQUIRE file contents from the listing.

EXPAND

Include the expansion of each macro call in the listing file. For an example of a macro expansion, see Section 3.2.4.3.

NOEXPAND

Do not include the expansion of macros.

TRACE

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 3.2.4.4.

NOTRACE

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.

OBJECT

Produce the object part of the output listing.

NOOBJECT

Suppress the object part of the output listing.

ASSEMBLY

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.

NOASSEMBLY

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.

Each listing switch is described and illustrated in Section 3.2.2 in connection with the discussion of the output listing produced by a BLISS compilation. Understanding the purpose of these listing switches requires knowledge of the format and purpose of the output listing, as discussed in that section.

2.4.7 Reference Switches

The reference switches allow a cross-reference listing to be included with the compiler listing. Further, a master cross-reference data file can be created (refer to Section 2.3) to produce a master

cross-reference listing (refer to master cross-reference utility program BCREF in Section 9.3). Some examples of using the reference switches are as follows:

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

  ```
  *,ALPHA=ALPHA/LIST/CREF
  ```

  The compiler produces list file ALPHA.LST 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:

  ```
  *,ALPHA=ALPHA/LIST:(NOSOURCE,NOOBJECT)/CREF
  ```

  The compiler produces list file ALPHA.LST, which contains only the cross-reference listing.

- To create only a master cross-reference data file, use the master cross-reference file specification:

  ```
  *,,ALPHA=ALPHA
  ```

  The compiler produces master cross-reference data file ALPHA.CRF.

- To produce a compiler listing that includes a cross-reference listing and a master cross-reference data file, use the following:

  ```
  *,ALPHA,ALPHA=ALPHA/CREF
  ```

  The compiler produces list file ALPHA.LST, to which a cross-reference listing is appended, and master cross-reference data file ALPHA.CRF.

- 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:

  ```
  *,ALPHA=ALPHA/CREF:MULTIPLE
  ```

  The compiler produces list file ALPHA.LST, to which a cross-reference listing is appended that includes multiple references to the same symbol.

2.4.7.1 Syntax - Reference qualifier syntax is defined as follows:

reference-switch	/CREF{:{ (reference-value ,...) }|:{ reference-value }}
reference-value	{ MULTIPLE | NOMULTIPLE }

2.4.7.2 Defaults - In the absence of an explicit choice of reference value, the following value is assumed by default:

NOMULTIPLE

2.4.7.3 Semantics - The /CREF switch indicates that cross-references are to be included in the listing and that zero or one reference-value will be given for the compilation. The reference-value have 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.

2.4.8 Environment Switches

Environment switches are used to specify the processor model and the operating system of the target system for which code is to be generated. Some examples of using environment switches are as follows:

- To generate code that uses instructions available only on a KL10 processor, use the following command line:

  ```
  *ALPHA=ALPHA/KL10
  ```

  The compiler reads the source from ALPHA.BLI and creates object file ALPHA.REL; this file makes use of KL10 instructions, such as ADJSP (which makes stack adjustments) and EXTEND (which implements various character-handling functions).

- To generate code that makes calls on the TOPS-20 monitor using JSYS instructions, use the following command line:

  ```
  *ALPHA=ALPHA/TOPS20
  ```

  If ALPHA contains the main routine of the program, the compiler generates a RESET% JSYS before the call to the main routine and a HALTF% JSYS immediately after the call to the main routine.

2.4.8.1 Syntax - Environment switch syntax is:

```
environment-switch   {/KA10 | /KI10 | /KL10 | /KS10}
                     {/TOPS10  | /TOPS20           }
```

2.4.8.2 Defaults - If no environment switches are specified, the following are assumed:

```
/KA10      /TOPS10
```

2.4.8.3 Semantics - Environment switches identify the target system for which code is being generated. Environment switches have the following meanings:

/KA10	Generate code that uses only the KA10 instruction set; this code executes on all processing models.
/KI10	Generate code that uses only the KI10 instruction set; this code executes on KI10 and KL10 processor models.
/KL10	Generate code that uses the KL10 instruction set; this code executes on KL10 and KS10 processor models.
/KS10	Generate code that uses the KS10 instruction set; this code executes on a KS10 processor models.
/TOPS10	Generate code that makes calls to the TOPS-10 monitor.
/TOPS20	Generate code that makes calls to the TOPS-20 monitor.

NOTE

A compiler-state-function is a lexical-function that expands to a numeric-literal of 1 or 0 during compilation to indicate whether a certain condition exists. The %SWITCHES lexical-function can be tested during compilation to determine the setting of one or more environment switches. For example, the following command line causes the %SWITCHES function to return the indicated numeric-literal:

```
*ALPHA=ALPHA/KL10/TOPS10

%SWITCHES(KA10)    - 0
%SWITCHES(KI10)    - 0
%SWITCHES(KL10)    - 1
%SWITCHES(KS19)    - 0
%SWITCHES(TOPS10)  - 1
%SWITCHES(TOPS20)  - 0
```

For additional information, refer to "Module-Switches" in the BLISS Language Guide.

2.4.9 Placement of Switches

Some directions can be given to the compiler either by command line switches or by switch settings contained in the module being compiled. The command line switch name is in some cases the same as the switch name contained in the module (module switches and SWITCHES declaration) and in other cases similar but not identical. The names for the common switches are given in Table 2-1.

2.4.10 Switches and Default Settings

Command-line switches alter default settings assumed for module switches. A switch setting in the module head overrides the corresponding switch given in the command line. A switch setting for a switches-declaration overrides the setting given in the module head.

Suppose you are compiling two programs. The first program ALPHA.BLI has a module switch CODE. The second program BETA has no switches. The bliss-command-line is as follows:

```
*=ALPHA,BETA/NOCODE
```

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

Table 2-1: Command Line, Module Switch, and SWITCH Names on TOPS-10

Command Line Name	Module Switch Name	SWITCHES Name
/CHECK	n/a	n/a
/CODE	CODE	n/a
/CREF	n/a	n/a
/DEBUG	DEBUG	n/a
/ERRS	ERRS	ERRS
/LIST:ASSEMBLY	LIST(ASSEMBLY)	LIST(ASSEMBLY)
/LIST:BINARY	LIST(BINARY)	LIST(BINARY)
/LIST:COMMENTARY	LIST(COMMENTARY)	LIST(COMMENTARY)
/LIST:EXPAND	LIST(EXPAND)	LIST(EXPAND)
/LIST:LIBRARY	LIST(LIBRARY)	LIST(LIBRARY)
/LIST:OBJECT	LIST(OBJECT)	LIST(OBJECT)
/LIST:REQUIRE	LIST(REQUIRE)	LIST(REQUIRE)
/LIST:SOURCE	LIST(SOURCE)	LIST(SOURCE)
/LIST:SYMBOLIC	LIST(SYMBOLIC)	LIST(SYMBOLIC)
/LIST:TRACE	LIST(TRACE)	LIST(TRACE)
/OPTLEVEL:n	OPTLEVEL:n	n/a
/SAFE	SAFE	SAFE
/UNAMES	UNAMES	UNAMES
/ZIP	ZIP	ZIP

n/a (not applicable) indicates that no corresponding switch exists.

2.4.11 Positive and Negative Forms of Switches

In general, two forms of a switch 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.

The positive and negative forms of a switch are mutually exclusive; only one form for any switch should be given in a bliss-command-line.

2.4.12 Abbreviations

The command switch names and value names can be abbreviated as long as SCAN can recognize the command unambiguously. This is true both for switches that can take values and switches that take no value.

2.5 SPECIAL FEATURES

2.5.1 Indirect Files

An indirect file is a file referenced within a BLISS command line; it is used to complete a BLISS command. The indirect file may contain a complete or partial BLISS command line: for example, filenames and switch settings. You reference the file by specifying an "at sign" (@) followed by the file-spec, the contents of which expands and complete the command line.

For example, assume the file MYPROG.CCL contains the following command lines:

```
LIB.L36=LIB1,LIB2,LIB3/LIBRARY
MYPROG=MYPROG
```

and that you issue the following command, using the indirect file MYPROG.CCL to specify command lines to be read:

```
*@MYPROG.CCL
```

The compiler produces library file LIB.L36 from the concatenation of source files LIB1, LIB2, and LIB3, compiles indirect file MYPROG, produces object file MYPROG.REL, and prompts again with an asterisk.

2.5.2 Option File

An option file named DSK:SWITCH.INI may reside in your logged in disk area; in it, you can set switches for use with various programs. These switches allow you to override system defaults for individual programs. The BLISS compiler is such a program.

The syntax of the lines in the option file for BLISS is as follows:

```
BLISS switch ...
```

or

```
BLISS:option-name switch
```

where option-name is a 1- to 6-character name.

For example, suppose that option file SWITCH.INI contains the lines:

```
BLISS /STATISTICS/DEBUG
BLIS16 /STATISTICS/DEBUG
BLISS:TERM /LIST:(NOBINARY,NOCOMMENTARY)/NOHEADER
```

The following command would cause the first line to be read; this sets the STATISTICS and DEBUG switches to be on as a default.

```
*ALPHA=ALPHA
```

The compiler compiles ALPHA, produces debug code, and prints routine names and sizes on the terminal.

Note that the BLISS-36 compiler looks for BLISS, while the BLISS-16 compiler looks for BLIS16.

To suppress debug code, type:

```
*ALPHA=ALPHA/NODEBUG
```

To suppress both statistics and debug code, type:

```
*ALPHA=ALPHA/NOSTATISTICS/NODEBUG
```

or

```
*ALPHA=ALPHA/NOOPTION
```

Specifying the /NOOPTION switch prevents the compiler from reading the SWITCH.INI file.

Given the following command, the compiler ignores the first BLISS line in the option file and reads only the second:

```
*,TTY:=ALPHA/OPTIONS:TERM
```

The compiler compiles ALPHA and sends the listing to the terminal. The listing would contain no binary, commentary, or page headers.

CHAPTER 3

COMPILER OUTPUT

This chapter discusses compiler output, starting with terminal output, followed by list file considerations, and finally error messages.

The input to a BLISS compilation is a BLISS program. 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 discussed in Chapter 12 of the BLISS Language Guide.

```
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);
    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.

3.1 TERMINAL OUTPUT

The compiler produces three kinds of information on the terminal: error messages, statistics, and a compilation summary. You can request or suppress error messages and statistics by using a /ERRS or /STATISTICS terminal-switch in the command line. (Refer to Section 1.) By default, error messages are reported during compilation, but statistics are suppressed. A compilation summary is always produced on the terminal when the compilation ends.

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 data words associated with that declaration. The compilation summary gives the number of warning and error messages, the number of words of code and data used by the program, the run time and the elapsed time required for the compilation, the number of lines and lexemes processed per CPU minute, and the number of pages of 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:

```
; Compilation Complete
```

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

```
; Library Precompilation Complete
```

Consider the terminal output for the sample module TESTFACT contained in the file MYPROG.BLI. To obtain all three kinds of information, the module is compiled by one of the following bliss-command-lines:

```
=>20
BLISS>MYPROG/STATISTICS

=>10
*MYPROG=MYPROG/STATISTICS
```

The /STATISTICS switch is used so that all three 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 9
RFACT 14
MAINPROG 9
.MAIN. 16
; Information:  0
; Warnings:     2
; Errors:       0
; Size:         48 code +  2050  data words
; Run time:        00:00.6
; Elapsed time:    00:01.0
; Lines/CPU Min:    3356
; Lexemes/CPU-Min: 13426
; Memory used:  3 pages
; Compilation Complete
```

This terminal output for compiling MYPROG includes two warnings, which are described in the following sections. Statistics follow the warnings and show the number of data words required for each routine. The example module TESTFACT contains three routine declarations: IFACT, RFACT, and MAINPROG. IFACT uses 9 words; RFACT uses 14 words; MAINPROG uses 9 words. Each main program is called by a small predefined routine (.MAIN.), which is called by the operating system; it requires 16 words.

The compilation summary shows that the compilation of TESTFACT required 0.6 second of processor time and 1.0 second of elapsed time to compile; moreover, 3356 source lines comprising 13426 lexemes were processed per CPU minute. The compilation required three pages of memory, exclusive of the memory required for the compiler itself.

3.2 OUTPUT LISTING

The output listing produced as a result of a BLISS compilation can be output on any suitable display device. It consists of source listings (including any error messages), optional object listings (as specified through command-line switches), 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 3-1).

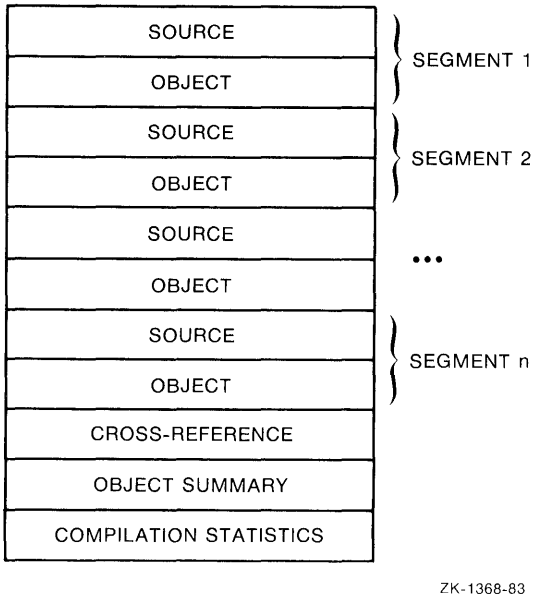


Figure 3-1: Compiler Output Listing Sequence

You can suppress both the source and the object parts of a routine segment, and change the format of the object part, by the inclusion of switches in the module or 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 high and low segment length 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 Appendix F). Only the first routine segment of that module is used here. 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 for that routine segment in detail.

3.2.1 Listing Header

Listing headers consist of two lines; each line consists of three fields separated by at least one column. The first field contains information in columns 1 through 15; the second extends from columns 17 through 63; the last extends from columns 65 through 132. The contents of each field are left-justified within the field. The listing header format appears in Figure 3-2.

The listing header format appears as follows:

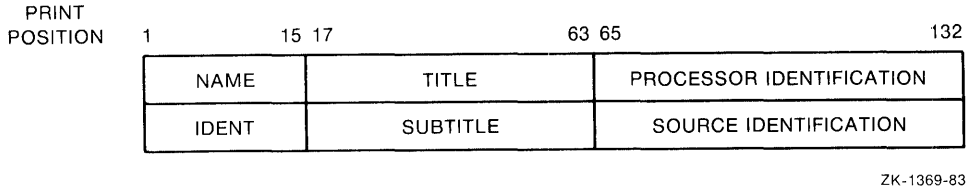


Figure 3-2: Listing Header Format

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 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 includes 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 one greater than the number of page marks (form feeds) read from the source.

3.2.2 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 while the 16-character string does not. The 16-character preface string has the general form:

```
;byznnnnbnnbbbbb
```

The 24-character preface string has the general form:

```
;xxxxxbbyznnnnbnnbbbbbbb
```

Table 3-1 describes the components of each string. (An asterisk denotes the components and columns of the 24-character string.)

For example, consider the following line from the BLISS input:

```
RESULT = 1;
```

If the above declaration is the fourteenth line in the compilation, the output listing for that line appears 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 of 02300, the output listing for the line would be:

```
;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 contained within a macro declaration, then the output listing includes an M, as follows:

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

The y item in 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. However, you can find the beginning of the unterminated macro by locating the point at which the M code first appeared in the y field before the runaway.

Table 3-1: Format of Preface String in Source Listing

Item	Column	Meaning
;	1	The comment character; used to comment out the source line so that the output listing can be assembled by the PDP-10 MACRO assembler.
xxxxx* or b	2-6* or 2	The line number, if the file contains line sequence numbers; otherwise, one blank column.
bb*	7-8*	Blanks
y	9* or 3	A code that indicates the lexical processing level of the compiler. The codes that can appear in this column are described below: **Code** — **Meaning** C — Embedded comment, that is, text within %(...)%. D — Default lexeme stream for a keyword macro formal. L — Parameter list of a lexical function. M — Body of a macro definition. P — Parameter list of a macro call. U — Source text which is discarded by an unsatisfied lexical condition. If more than one such code applies (for example, an embedded comment nested within a macro body), the "innermost" code is printed.
z	10* or 4	If the line comes from a file specified in a REQUIRE declaration, the code "R"; otherwise, a blank.
nnnn	11-14* or 5-8	The BLISS line sequence number, beginning with 0001, 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 reflects the depth of the code within each block structure.
bbbbbbb*	18-24*	Blanks
bbbbb	12-16	Blanks

An example of the source listing for the first segment of the module TESTFACT, which uses the 16-character preface string, appears below.

```
        .
        .
        .
;   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;
                .
                .
                .
```

Following the three heading lines, which have been omitted in this example, the source of the module TESTFACT is reproduced. The 16-character preface string begins with a semicolon (;). Since the input file that contains the module TESTFACT does not have sequence numbers, column 2 of the source listing is blank. Columns 3 and 4 are blank, because the lexical processing level is normal and the material is not from a REQUIRE file. Line numbers generated by the compiler begin in column 5. (See Figure 3-9 for a complete listing.)

Both error messages are reported as part of the source listing. Section 3.4 contains a discussion of error messages in general and of the meaning of these errors in particular.

3.2.3 Object Listing

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

However, although 16 different forms of listings are theoretically possible, in practice only a few combinations of format-options are meaningful.

The following combinations of the format options are reasonable:

```
ASSEMBLER { SYMBOLIC   }  COMMENTARY { BINARY   } { UNAMES   }
          { NOSYMBOLIC }             { NOBINARY } { NOUNAMES }
```

The commentary field requires little space and provides useful information about source line numbers, so that, currently, you have no need for the NOCOMMENTARY switch. Also, there is little reason to specify the NOASSEMBLER switch, since its only effect is to suppress the macro END statement.

The question of whether to have the binary appear on the listing is one of personal preference. However, it may be useful for debugging purposes.

The compiler produces the following information for each field.

ASSEMBLER field — Instructions in assembler form. For example:

```
MOVEI  AC16,  1
```

SYMBOLIC field — Operands using symbolic source names. For example:

```
I , 1
```

BINARY field — Octal equivalent of instructions and data to facilitate debugging. The octal instructions appear as much as possible in the same format as that produced by the MACRO assembler.

The following codes may be appended to octal values in the binary field to provide information about relocation of quantities:

Code	Meaning
Blank	Absolute quantity (no linker action)
V	Forward relocatable
#	Relocated by POLISH expression
'	Relocated relative to PSECT
*	Relocated relative to external symbol

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 instruction generated first are left blank.

The partial printout of a default object listing appearing in Figure 3-3 illustrates the object part of the routine segment. The command line

```
=>20
BLISS>TESTFACT/LIST

=>10
*TESTFA,TESTFA=TESTFA
```

generated the listing, in which the assembler field appears first, followed by the symbolic field, the binary field, and the commentary field. Note that the default switch settings in effect are ASSEMBLER, SYMBOLIC, COMMENTARY, BINARY, and NOUNAMES.

```
                        .
                        .
                        .
                TITLE   TESTFACT
                TWOSEG
                .REQUEST  SYS:B36OTZ.REL
                RELOC   0                       ;                       000000'
A:              BLOCK   1                       ;                       000000'
B:              BLOCK   1                       ;                       000001'
                EXTERN  REULT
AC0=            0
AC1=            1
AC2=            2
AC3=            3
AC4=            4
AC5=            5
AC6=            6
AC7=            7
AC10=           10
AC11=           11
AC12=           12
AC13=           13
AC14=           14
FP=             15
AC16=           16
SP=             17
                RELOC   400000                  ;                       400000'
IFACT:          MOVEI   AC1,1                   ; RESULT,1              400000' 201 01 0 00 000001
0012
                MOVEI   AC2,1                   ; I,1                   400001' 201 02 0 00 000001
0013
                JRST    L.2                     ; L.2                   400002' 254 00 0 00 400005'
L.1:            MOVE    AC1,REULT               ; RESULT,REULT          400003' 200 01 0 00 000000*
0014
                IMUL    AC1,AC2                 ; RESULT,I              400004' 220 01 0 00 000002
L.2:            ADDI    AC2,1                   ; I,1                   400005' 271 02 0 00 000001
0013
                CAMG    AC2,-1(SP)              ; I,N                   400006' 317 02 0 17 777777
                JRST    L.1                     ; L.1                   400007' 254 00 0 00 400003'
                POPJ    SP,                     ; SP,                   400010' 263 17 0 00 000000
0008
; Routine Size:  9 words
                        .
                        .
                        .
```

Figure 3-3: Default Object Listing Example

3.2.4 Source Part Options

The following sections contain more output listings to illustrate different options for the source part of the list file. To illustrate different forms, the sample program TESTFACT has to be made more interesting, along the lines given in the following paragraphs.

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 obtained. The next step is to produce a new module TEST, which uses the factorial routine to take combinations according to the following formula for obtaining the number of combinations of m items taken n at a time:

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

where m! is the notation for the factorial of m.

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 for obtaining the combinations, namely:

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

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

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

```
=>20
BLISS>COMBN/LIBRARY

=>10
*COMBN=COMBN/LIBRARY
```

Finally, include some test combinations.

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

3.2.4.1 Default Source Listing - The command line

```
=>20
BLISS>TEST/LIST/NOCODE

=>10
*TEST,TEST=TEST/NOCODE
```

generated the output listing in Figure 3-4 for the module TEST. Note 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 0014 are used for this purpose.

3.2.4.2 Listing with LIBRARY/REQUIRE Information - The command line

```
=>20
BLISS>TEST/NOCODE/LIST/FORMAT:(LIBRARY,REQUIRE)

=>10
*TEST,TEST=TEST/NOCODE/LIST:(LIBRARY,REQUIRE)
```

generated the output listing in Figure 3-5, which contains information from the LIBRARY and REQUIRE files. The LIBRARY file is identified following line 0009 and the first use of a name from that library is noted following line 0018. The contents of the REQUIRE file are given in lines 0011 through 0014.

3.2.4.3 Listing with Macro Expansions - The command line

```
=>20
BLISS>TEST/NOCODE/LIST/FORMAT:EXPAND

=>10
*TEST,TEST=TEST/NOCODE/LIST:EXPAND
```

generated the output listing in Figure 3-6 to illustrate macro expansions, which follow lines 0018 and 0019. Note 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. The last line of the macro expansion, therefore, is the fully expanded form.

3.2.4.4 Listing with Macro Tracing - The command line

```
=>20
BLISS>TEST/NOCODE/LIST/FORMAT:TRACE

=>10
*TEST,TEST=TEST/NOCODE/LIST:TRACE
```

produced the output listing in Figure 3-7, which contains macro tracing and macro expansions. The macro trace gives information about parameter binding in addition to the expansion information.

```
                    .
                    .
                    .
;   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  3           A = COMBINATIONS (3, 2);
;   0021  2           B = COMBINATIONS (6, 4);
;   0022  1           END;
;   0023  1
;   0024  1       END
;   0025  0       ELUDOM
;                                  LIBRARY STATISTICS
;
;                                              -------- Symbols --------     Blocks          Processing
;         File                                 Total    Loaded   Percent       Read            Time
;
;                                                  2         2       100          4            00:00.0
; Run Time:          00:00.3
; Elapsed Time:      00:00.8
; Lines/CPU Min:     5952
; Lexemes/CPU-Min: 49523
; Memory Used:    3 pages
; Compilation Complete
```

Figure 3-4: Default Source Listing Example

```
                    .
                    .
                    .
;    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  produced by TOPS-20 Bliss-36 3A(177) on  3-May-1983 16:10:24
;    0010  1       REQUIRE 'RFACT' ;
;   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;
;    0017  1
;    0018  1       ROUTINE MAINPROG =
;    0019  2           BEGIN
;    0020  2           A = COMBINATIONS (3, 2);
;; Loaded symbol COMBINATIONS from library
;; Loaded symbol COMB from library
;    0021  2           B = COMBINATIONS (6, 4);
;    0022  1           END;
;    0023  1
;    0024  1       END
;    0025  0       ELUDOM
;                                LIBRARY STATISTICS
;
;                                                 -------- Symbols --------     Blocks          Processing
;         File                                    Total    Loaded   Percent       Read            Time
;
;                                                    2         2       100          4             00:00.0
; Run Time:          00:00.3
; Elapsed Time:      00:01.3
; Lines/CPU Min:     4823
; Lexemes/CPU-Min: 40128
; Memory Used:    3 pages
; Compilation Complete
```

Figure 3-5: Output Listing with Library and Require File Data

```
                    .
                    .
                    .
;    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 ( ) / ( FACT ( - ) * FACT ( ) )
;;     [COMBINATIONS]=  ( IF LSS THEN ERROR ( ) ELSE FACT ( ) / ( FACT ( - ) * FACT ( ) ) )
;    0021  2            B = COMBINATIONS (6, 4);
;;           [COMB]=  FACT ( ) / ( FACT ( - ) * FACT ( ) )
;;     [COMBINATIONS]=  ( IF LSS THEN ERROR ( ) ELSE FACT ( ) / ( FACT ( - ) * FACT ( ) ) )
;    0022  1            END;
;    0023  1
;    0024  1        END
;    0025  0        ELUDOM
;                                  LIBRARY STATISTICS
;
;                                              -------- Symbols --------     Blocks          Processing
;         File                                 Total    Loaded   Percent       Read            Time
;
;                                                  2         2       100          4            00:00.0
; Run Time:          00:00.3
; Elapsed Time:      00:01.2
; Lines/CPU Min:     4983
; Lexemes/CPU-Min: 41461
; Memory Used:    3 pages
; Compilation Complete
```

Figure 3-6: Output Listing with Macro Expansion Data

```
                .
                .
                .
;   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 ( ) / ( FACT ( - ) * FACT ( ) )
;;    [COMBINATIONS]=  ( IF LSS THEN ERROR ( ) ELSE FACT ( ) / ( FACT ( - ) * FACT ( ) ) )
;   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 ( ) / ( FACT ( - ) * FACT ( ) )
;;    [COMBINATIONS]=  ( IF LSS THEN ERROR ( ) ELSE FACT ( ) / ( FACT ( - ) * FACT ( ) ) )
;   0022  1          END;
;   0023  1
;   0024  1      END
;   0025  0      ELUDOM
;                              LIBRARY STATISTICS
;
;                                        -------- Symbols --------    Blocks          Processing
;       File                             Total    Loaded   Percent     Read            Time
;
;                                            2         2       100        4            00:00.0
; Run Time:          00:00.3
; Elapsed Time:      00:01.2
; Lines/CPU Min:     4687
; Lexemes/CPU-Min: 39000
; Memory Used:    3 pages
; Compilation Complete
```

Figure 3-7: Output Listing with Macro Expansion and Tracing Data

3.3 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 appears as the last module-specific item in the listing file, before the compilation summary and before any subsequent module data.

3.3.1 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 ...
--------------- -------- ----- -----------------
```

3.3.2 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 symbol 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 3-2.

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, or Map 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.

Table 3-2: Symbol Type Abbreviations

Meaning	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
Stacklocal	Stackloc
Structure Formal	StruForm
Name which is not bound	Unbound
Undeclare	Undeclar

As an example of the use of 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. Since 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 intial line number and its 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 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 two-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:

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

Note that since 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:

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

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 is flagged for B.

3.3.3 Output Listing with Cross-Reference Listing

The listing in Figure 3-8 includes a cross-reference listing that was produced by compiling module TEST with the following options:

```
=>20
BLISS>TEST/FORMAT:REQUIRE/LIST/CROSS-REFERENCE

=>10
*TEST,TEST=TEST/LIST:(REQUIRE)/CREF
```

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, and a flags legend, which describes the codes used in the Referenced field.

```
     .
     .  (header)
     .
;    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' ;
;   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;
;    0017  1
;    0018  1     ROUTINE MAINPROG : NOVALUE =
;    0019  2         BEGIN
;    0020  2         A = COMBINATIONS (3, 2);
;    0021  2         B = COMBINATIONS (6, 4);
;    0022  1         END;
;    0023  1
;    0024  1     END
;    0025  0     ELUDOM
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    14       20       21
MAINPROG      Routine     18
N             RoutForm    11    12       14
                 CROSS REFERENCE MAP
Line #       Event         File ...
-----  ---------------  ---------
    1  Source (start)   PS:<DIRECTORY>TEST.B36.3
    1  Module TEST
    9  Library #1
   11  Require (start)  PS:<DIRECTORY>RFACT.R36.3
   16  Require (end)
   25  Eludom TEST
        KEY TO REFERENCE TYPE FLAGS
            .   Fetch
            =   Store
            c   Routine call
            a   Address use
               Indirect use
            f   Forward or forward routine declaration
            m   Map declaration
            h   Condition handler enabling
     .
     .
     .
```

Figure 3-8: Output Listing with Cross-Reference Listing Included

3.4 COMPILATION SUMMARY

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

- The routine size and psect-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
- Number of warnings and errors (omitted if no warnings or errors exist)
- 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

3.5 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. Both the warning and the fatal errors 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 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 identification 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 messages 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  L2:0018  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 error, while omitting the close parenthesis (that is, (N =) caused one fatal error. With the absence of a blank separator, 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 lines; 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 remain unscanned. Also, the scan sometimes causes genuine errors to be missed or 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 line classifies the error as a nonfatal by the string WARN and gives the error 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 Figure 3-9. 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.

```
    .
    .  (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 :NOVALUE =
;   0026  2         BEGIN
;   0027  2         A = IFACT(5);
;   0028  2         B = RFACT(5);
;   0029  1         END;
;   0030  1
;   0031  1     END
;   0032  0     ELUDOM
; Information:  0
; Warnings:     4
; Errors:       1
    .
    .
    .
```

Figure 3-9: Error Messages in Source Listing Example

CHAPTER 4

LINKING, EXECUTING, AND DEBUGGING

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

4.1 LINKING

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

To invoke LINK, use the following command:

```
=>20      @LINK
=>10      .R LINK
```

LINK builds an executable image of your program. The /GO switch causes LINK to terminate. To create a file from the image, issue the SAVE or NSAVE command as follows:

```
=>20      @SAVE ALPHA
=>10      .NSAVE ALPHA
```

In both cases a single image file (ALPHA.EXE) is saved. (Other versions of SAVE are available on TOPS-10. Consult the DECsystem-10 Operating System Commands Manual for details.)

The following command causes ALPHA to be executed:

```
=>20      @RUN ALPHA
=>10      .RUN ALPHA
```

Some examples of linking are:

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

  ```
  *ALPHA
  */GO
  ```

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

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

  ```
  *ALPHA
  *BETA
  *GAMMA
  */GO
  ```

In response to this command, the linker combines the object module in the file ALPHA.REL with the object module in the file BETA.REL and the object module in the file GAMMA.REL to produce a single executable image.

Linking a program compiled for extended addressing requires special coding considerations and link commands. Due to linker restrictions, the compiler generates PSECTED code for all programs using the extended addressing option (/EXTENDED). Such programs are then linked into a specified nonzero section; however, if the compiled program uses /EXTENDED:SECTION-INDEPENDENT code, which allows loading into any section, section zero must be specified at link time (refer to Section 6.4 for examples).

The default PSECTS generated are: OWN, $GLOBAL$, $PLIT$, and $CODE$, the origins of which are set at link time.

The following example shows how a program using the extended addressing option is linked and loaded into section one.

```
@LINK
*ALPHA/SAV =
*/SET:$OWN$:1200000
*/SET:$GLOBAL$:1300000
*/SET:$PLIT$:1400000
*/SET:$CODE$:1500000
*/SYMSEG:PSECT:$OWN$/PVBLOCK:PSECT:$CODE$
*ALPHA
*/GO
```

Note that when an extended addressing program is loaded, the placement of the program's data vector (PVBLOCK) and symbol table (SYMSEG) must be specified.

The link operation is described in detail in the LINK-10 Programmer's Reference Manual and the DECSYSTEM-20 LINK Reference Manual.

4.1.1 Syntax

The following syntax for linking a BLISS program works under both TOPS-10 and TOPS-20, and is sufficient for a simple BLISS program. The full description of LINK commands can be found in the appropriate link manual.

link-command	{ test-line / nothing } object-line ... exit-line
test-line	/TEST
object-line	object-spec
exit-line	/GO

A carriage return is used to terminate each test-line, object-line, and exit-line.

4.1.2 Defaults

If a file type is not included in the object-spec, the file type .REL is assumed.

4.1.3 Semantics

LINK reads the object modules contained in each object file named in the link command to create a linked, executable image. The name of the executable image is specified in the SAVE command (see Section 4.1).

There are a number of minor differences between LINK under TOPS-10 and TOPS-20. Under TOPS-10, /TEST causes DDT to be loaded at the end of the program's low segment, while under TOPS-20, DDT gets "mapped" at page 770. Under TOPS-10, a run-time symbol table is omitted by default, unless /TEST is specified, while under TOPS-20, symbols are included for all nonoverlayed images, unless /NOLOCAL is specified.

The /MAXCOR and /SAVE switches are TOPS-10 specific.

Under TOPS-20, when accessing files from a directory other than that currently connected to, either use a programmer-project number or define a logical name for the new directory. The TRANSLATE and DEFINE commands can be used for this purpose.

NOTE

For additional information related to linking BLISS-36 modules with modules written in assembly language or BLISS-10, refer to Appendices G and H.

4.2 EXECUTING

To run your program, use the executable image produced as a result of the link operation, in a RUN command, as shown above.

Your program, ALPHA, then executes. Any input or output in your program takes place. If your program is correct, it runs to completion and returns to the command processor. The command processor then prompts for another command.

4.3 DEBUGGING

If your program has problems or if you want to examine some data within your program, you can use the SIX12 debugger. Using the switch /DEBUG in the compilation operation tells LINK to load SIX12 and to pass the symbol tables from the REL files to the image file. The executable image formed as a result contains SIX12, and when the image is executed, control first goes to SIX12 instead of your program. You can then examine and deposit values in storage, set breakpoints, call routines, or do any of a number of other debug or test operations. The SIX12 debugger is described basically in SIX12.HLP (a help file distributed with the compiler), while a more comprehensive description can be found in the SIX12.DOC file, distributed with the compiler.

4.3.1 Debug Example

As an example of using the debug facility, consider the testing of the program MYPROG. Errors detected in MYPROG in Section 3 are corrected, and then MYPROG is compiled, linked with SIX12, and run.

Assume that MYPROG has been successfully compiled with the /DEBUG switch.

```
@LINK
*/TEST
*MYPROG
*/GO
EXIT
@SAVE MYPROG
MYPROG SAVED
@RUN MYPROG
SIX12 V8-4 (TOPS-20 I/O) FOR BLISS-36
&IFACT(5)

          170     = =    .STACK + 26

&RFACT(5)

          170     = =    .STACK + 26

&GO
@
```

In the above example, the first step is to invoke LINK, which prompts with the asterisk (*). Next, the /TEST switch is specified to indicate that a debugging version of a program is to be built. Then follows a sequence of object files that are to be included in the build. In this case there is only one: MYPROG. The final command given to LINK is /GO, which causes LINK to terminate and leave the image file in memory. The user saves MYPROG with the SAVE command and executes it with the RUN command.

In this case, SIX12 gets control when MYPROG is run, and informs you as to which version of SIX12 this is, and whether TOPS-10 or TOPS-20 I/O is being used. SIX12 prompts with an ampersand (&). The user has requested that SIX12 call routine IFACT and routine RFACT with actual parameter 5 in each case. In each case, the resultant value is 120 (170 octal).

The default radix that SIX12 uses is octal. To specify a decimal integer to SIX12, precede it by a hash mark (#).

4.3.2 Other SIX12 Commands

Some commonly used SIX12 commands are now described. The basic syntax of SIX12 is similar to BLISS. Therefore, to examine GLOBAL or OWN data (for example, A) type:

```
&.A
```

To examine the current stack of routine invocations, type:

```
&CALLS
```

To change the value of an OWN or GLOBAL variable A, type:

```
&A=new-value
```

To set a breakpoint on entry to a routine, type:

```
&BREAK routine-name
```

To set a breakpoint on routine exit, type:

```
&ABREAK routine-name
```

The commands DBREAK and DABREAK clear the above two breakpoints, respectively.

To begin execution, type:

```
&GO
```

As the above example indicates, to call a routine, type:

```
&routine-name(actual-parameter-list)
```

To invoke DDT, type:

```
&DDT
```

To return from DDT to SIX12, type the following to DDT:

```
&SIXRET$X   (where $ is altmode or escape)
```

To examine the n'th actual parameter of the current routine when stopped at its entry or exit, type:

```
&n%A
```

SIX12 prints out the values of the actual parameters when the CALLS command is given or when a breakpoint has been reached.

CHAPTER 5

MACHINE-SPECIFIC FUNCTIONS

Machine-specific functions allow you to perform specialized operations within the BLISS language. A machine-specific function call is similar to a BLISS routine call. It requires parameters and returns a value.

The compilation of a machine-specific function results in the generation of some inline code, often a single instruction, rather than a call to an external routine. The one exception to this is the Convert Double to Floating (CVTDF) function.

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, optimization procedures generate a different machine instruction from the one specified in the call.

Machine-specific functions in BLISS-36 are divided into four categories, as illustrated in Table 5-1. A separate description of each function appears below. For a more detailed discussion, consult the DECsystem-10/DECSYSTEM-20 Hardware Reference Manual.

5.1 GENERAL CONVENTIONS

The definitions of these functions require addresses, values, or register names as parameters, even though register names do not have values in BLISS-36.

All expressions can be run-time computable expressions, unless specified otherwise in the descriptions that follow.

In each description, the calling sequence is given first, followed by a description of the parameters. The actual semantics of the function are specified under "Return Value."

5.1.1 Machine Code Insertion Functions

Machine code insertion functions provide a means to specify a particular -10/-20 instruction to be executed. These functions are intended for highly specialized applications which cannot be otherwise directly specified in the language. The user of these functions is cautioned that no extensive analysis of the instruction is performed by the compiler; it is possible for the user to specify an instruction that interferes with compiler-generated instructions and results in an incorrect program. Therefore, avoid the use of these functions to perform an operation that can be expressed more directly in the language. In particular, user manipulation of the stack pointer is very risky and requires extreme care.

Table 5-1: Machine-Specific Functions

Logical Functions	
ASH	Arithmetically shift a value
FIRSTONE	Find the leftmost non-zero list in a value
LSH	Logically shift a value
ROT	Rotate a given value

Byte Manipulation Functions	
COPYII	Increment both source and destination byte pointers and copy a byte
COPYIN	Increment a source byte pointer and copy a byte
COPYNI	Increment a destination byte pointer and copy a byte
COPYNN	Copy a byte
DPB	Deposit a byte
INCP	Increment a byte pointer
LDB	Load a byte
POINT	Build a -10/-20 byte pointer
REPLACEI	Increment a byte pointer and store a byte
REPLACEN	Store a byte given a byte pointer
SCANI	Increment a byte pointer and fetch a byte
SCANN	Fetch a byte given a byte pointer

Arithmetic Functions	
ADDD	Add double operands
ADDF	Add floating operands
ADDG	Add float-G operands
DIVD	Divide double operands
DIVF	Divide floating operands
DIVG	Divide float-G operands
MULD	Multiply double operands
MULF	Multiply floating operands
MULG	Multiply float-G operands
SUBD	Subtract double operands
SUBF	Subtract floating operands
SUBG	Subtract float-G operands

Arithmetic Comparison Functions	
CMPD	Compare double operands
CMPF	Compare floating operands
CMPG	Compare float-G operands

Arithmetic Conversion Functions	
CVTDF	Convert double to floating
CVTDI	Convert double to integer
CVTFD	Convert floating to double
CVTFG	Convert floating to float-G
CVTFI	Convert floating to integer
CVTGF	Convert float-G to floating
CVTGI	Convert float-G to integer
CVTID	Convert integer to double
CVTIG	Convert integer to float-G
CVTIF	Convert integer to floating

(continued on next page)

Table 5-1 (Cont.): Machine-Specific Functions

Machine Code Insertion Functions	
MACHOP	Execute a -10/-20 instruction
MACHSKIP	Execute a -10/-20 instruction and record whether a skip occurred
System Interface Functions	
JSYS	Perform a TOPS-20 Monitor call
UUO	Perform a TOPS-10 Monitor call

5.1.2 Logical Functions

If the arguments to the functions defined in this section are compile-time constant expressions, the result is also a compile-time constant expression.

5.1.3 Arithmetic Functions

The arithmetic functions provide single-, double-, and extended double-precision floating-point operations. Note that where the parameters for these functions require the addresses of operands, registers may only be used for single-precision float operands (ADDF, SUBF, MULF, CMPF, CVTFx, and CVTxF) and integer operands (CVTIx and CVTxI).

5.1.4 System Interface Functions

BLISS provides a way to request that TOPS-10 and TOPS-20 system services be performed. For TOPS-10, the machine-specific function used to communicate this request for service is UUO. For native mode TOPS-20 programs, the JSYS linkage-type should be used. For compatibility mode programs on TOPS-20, that is, TOPS-10 programs running with the PA1050 Emulator, UUO is used.

Using both JSYS and UUO together is not recommended; the TOPS-10 Emulator PA1050 is not guaranteed to handle UUOs correctly while the program contains JSYS requests.

An example of using JSYS and UUO can be found in Section 9.6.

Information on specific system services can be found in the TOPS-20 Monitor Calls Reference Manual and the DECsystem-10 Monitor Calls Manual.

5.2 ADDD (ADD DOUBLE OPERANDS)

Calling Sequence:

```
ADDD (SRC1A, SRC2A, DSTA)
```

Parameters:

SRC1A	Address of a double-precision floating-point value used as the addend
SRC2A	Address of a double-precision floating-point value used as the augend
DSTA	Address where the sum of operand 1 and operand 2 is stored

Return Value:

NOVALUE

5.3 ADDF (ADD FLOATING OPERANDS)

Calling Sequence:

ADDF (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A	Address of a single-precision floating-point value used as the addend
SRC2A	Address of a single-precision floating-point value used as the augend
DSTA	Address where the sum of operand 1 and operand 2 is stored

Return Value:

NOVALUE

5.4 ADDG (ADD FLOAT-G OPERANDS)

Calling Sequence:

ADDG (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A	Address of an extended double-precision floating point value used as the addend
SRC2A	Address of an extended double-precision floating point value used as the augend
DSTA	Address where the sum of operand 1 and operand 2 is stored

Return Value:

NOVALUE

5.5 ASH (ARITHMETIC SHIFT)

Calling Sequence:

```
ASH(EXP,CEXP)
```

Parameters:

EXP	Value to be arithmetically shifted
CEXP	Number of bit positions to be shifted

Return Value:

If EXP is greater than zero, EXP is shifted left by CEXP bit positions; otherwise, EXP is shifted right CEXP bits. During a right shift, bits shifted out of bit 1 are replaced with a copy of the sign bit.

5.6 CMPD (COMPARE DOUBLE OPERANDS)

Calling Sequence:

```
CMPD (SRC1A, SRC2A)
```

Parameters:

SRC1A	Address of a double-precision floating-point value for comparison
SRC2A	Address of a double-precision floating-point value for comparision

Return Value:

-1	SRC1A less than SRC2A
0	SRC1A equal to SRC2A
1	SRC1A greater than SRC2A

5.7 CMPF (COMPARE FLOATING OPERANDS)

Calling Sequence:

```
CMPF (SRC1A, SRC2A)
```

Parameters:

SRC1A	Address of a single-precision floating-point value for comparison
SRC2A	Address of a single-precision floating-point value for comparison

Return Value:

-1	SRC1A less than SRC2A
0	SRC1A equal to SRC2A
1	SRC1A greater than SRC2A

5.8 CMPG (COMPARE FLOAT-G OPERANDS)

Calling Sequence:

```
CMPG (SRC1A, SRC2A)
```

Parameters:

SRC1A	Address of an extended double-precision floating-point value for comparison
SRC2A	Address of an extended double-precision floating-point value for comparison

Return Value:

-1	SRC1A less than SRC2A
0	SRC1A equal to SRC2A
1	SRC1A greater than SRC2A

5.9 COPYII, COPYIN, COPYNI, AND COPYNN (COPY A BYTE)

Calling Sequence:

```
COPYII(SAPI, DAPI)
COPYIN(SAPI, DAP)
COPYNI(SAP, DAPI)
COPYNN(SAP, DAP)
```

Parameters:

SAP	Address of a source byte pointer
SAPI	Address of a source byte pointer to be incremented
DAP	Address of a destination byte pointer
DAPI	Address of a destination byte pointer to be incremented

Return Value:

COPYNN copies the field defined by the first parameter into the field defined by the second parameter. COPYNI increments the second byte pointer and proceeds as for COPYNN. COPYIN increments the first byte pointer and proceeds as for COPYNN. COPYII increments both byte pointers and proceeds as for COPYNN.

In all cases the value returned is the value fetched.

5.10 CVTDF (CONVERT DOUBLE TO FLOATING)

Calling Sequence:

```
CVTDF(SRCA, DSTA)
```

Parameters:

SRCA	Address of a double-precision floating-point value for conversion
DSTA	Address where the single-precision floating-point conversion is stored

Return Value:

NOVALUE

5.11 CVTDI (CONVERT DOUBLE TO INTEGER)

Calling Sequence:

```
CVTDI (SRCA, DSTA)
```

Parameters:

SRCA	Address of a double-precision floating-point value for conversion
DSTA	Address where the integer conversion is stored

Return Value:

1	No integer overflow
0	Integer overflow

5.12 CVTFD (CONVERT FLOATING TO DOUBLE)

Calling Sequence:

```
CVTFD (SRCA, DSTA)
```

Parameters:

SRCA	Address of a single-precision floating-point value for conversion
DSTA	Address where the double-precision floating-point conversion is stored

Return Value:

NOVALUE

5.13 CVTFG (CONVERT FLOATING TO FLOAT-G)

Calling Sequence:

```
CVTFG (SRCA, DSTA)
```

Parameters:

SRCA	Address of a single-precision floating-point value for conversion
DSTA	Address where the extended double-precision floating-point conversion is stored

Return Value:

NOVALUE

5.14 CVTFI (CONVERT FLOATING TO INTEGER)

Calling Sequence:

CVTFI (SRCA, DSTA)

Parameters:

SRCA	Address of a single-precision floating-point value for conversion
DSTA	Address where the integer conversion is stored

Return Value:

1	No integer overflow
0	Integer overflow

5.15 CVTGF (CONVERT FLOAT-G TO FLOATING)

Calling Sequence:

CVTGF (SRCA, DSTA)

Parameters:

SRCA	Address of an extended double-precision floating-point value for conversion
DSTA	Address where the single-precision floating-point conversion is stored

Return Value:

1	No integer overflow
0	Integer overflow

5.16 CVTGI (CONVERT FLOAT-G TO INTEGER)

Calling Sequence:

CVTGI (SRCA, DSTA)

Parameters:

SRCA	Address of an extended double-precision floating-point value for conversion
DSTA	Address where the integer conversion is stored

Return Value:

1	No integer overflow
0	Integer overflow

5.17 CVTID (CONVERT INTEGER TO DOUBLE)

Calling Sequence:

```
CVTID (SRCA, DSTA)
```

Parameters:

SRCA	Address of an integer value for conversion
DSTA	Address where the double-precision floating-point conversion is stored

Return Value:

NOVALUE

5.18 CVTIF (CONVERT INTEGER TO FLOATING)

Calling Sequence:

```
CVTIF (SRCA, DSTA)
```

Parameters:

SRCA	Address of an integer value for conversion
DSTA	Address where the single-precision floating-point conversion is stored

Return Value:

NOVALUE

5.19 CVTIG (CONVERT INTEGER TO FLOAT-G)

Calling Sequence:

```
CVTIG (SRCA, DSTA)
```

Parameters:

SRCA	Address of an integer value for conversion
DSTA	Address where the extended double-precision floating-point conversion is stored

Return Value:

NOVALUE

5.20 DIVD (DIVIDE DOUBLE OPERANDS)

Calling Sequence:

DIVD (DIVSR, DIVID, QUOT)

Parameters:

DIVSR	Address of a double-precision floating-point value used as the divisor
DIVID	Address of a double-precision floating-point value used as the dividend
QUOT	Address where the quotient is stored

Return Value:

NOVALUE

5.21 DIVF (DIVIDE FLOATING OPERANDS)

Calling Sequence:

DIVF (DIVSR, DIVID, QUOT)

Parameters:

DIVSR	Address of a single-precision floating-point value used as the divisor
DIVID	Address of a single-precision floating-point value used as the dividend
QUOT	Address where the quotient is stored

Return Value:

NOVALUE

5.22 DIVG (DIVIDE FLOAT-G OPERANDS)

Calling Sequence:

DIVG (DIVSR, DIVID, QUOT)

Parameters:

DIVSR	Address of an extended double-precision floating-point value used as the divisor
DIVID	Address of an extended double-precision floating-point value used as the dividend
QUOT	Address where the quotient is stored

Return Value:

NOVALUE

5.23 FIRSTONE (FIND FIRST BIT)

Calling Sequence:

FIRSTONE(EXP)

Parameters:

EXP	The value to be examined

Return Value:

-1 if EXP is equal to zero; otherwise, the number of high-order zero bits to the left of the first one bit in EXP.

5.24 INCP (INCREMENT A BYTE POINTER)

Calling Sequence:

INCP(AP)

Parameters:

AP	Address of a byte pointer

Return Value:

Increment the byte pointer at location AP. Return 0.

5.25 JSYS (INVOKE A TOPS-20 SYSTEM SERVICE)

Calling Sequence:

JSYS (skips, number{ , regname , ... })

Parameters:

skips	A compile-time constant expression whose value is in the range -1 to 2. The value of this expression indicates the manner in which control may return following the JSYS instruction.
number	The low-order 18 bits of this value become the effective address of the JSYS instruction (must not be a register name nor a %REF function).
regname	The name of a register. Any number of register names (or none) may be given as the third through last parameters.

A JSYS instruction (opcode 104) with the specified effective address (and a zero accumulator field) is executed.

The value of the "skips" parameter must agree with the characteristics of the JSYS which is executed. It defines the manner in which control may return following the JSYS, and the value of the JSYS function.

If the JSYS is one which has skip returns, the skips parameter must be specified as 1 or 2, depending upon the number of alternate returns. If the JSYS is one which has a single return, the skips parameter must be specified as -1 or 0. If you desire a software interrupt on the occurrence of a JSYS error, the value 0 is selected. If you desire to handle the occurrence of a JSYS error at the call site by using the ERJMP facility, the value -1 is selected.

Note that the GOTO return possible by specifying a nonzero reparse dispatch address to the COMND JSYS is not supported by the JSYS machine- specific function.

Detailed specifications of the value of the "skips" parameter follow:

SKIPS Value	Interpretation
-1	Control always returns to the calling location plus 1. This location contains an ERJMP instruction. The value of the function is zero if an error occurred during execution of the JSYS, and 1 if no error occurred.
0	Control always returns to the calling location plus 1. (This location will not contain an ERJMP instruction.) The value of the function is zero.
1	Control returns either to the calling location plus 1 or to the calling location plus 2. The value of the function is zero if control returns to the calling location plus 1, and 1 if control returns to the calling location plus 2.
2	As for 1, except that control may also return to the calling location plus 3, in which case the value of the function is 2.

The usage of accumulators by the JSYS to receive parameters and to return values is specified by the list of "regname" parameters. Each parameter is the name of a register data segment which has been allocated by the programmer to a specific machine register by means of a declaration similar to the following:

```
REGISTER
    AC1=1;
```

Each accumulator which may be referenced by the JSYS, either as an input or as an output, must be identified by this means. The compiler assumes that accumulators which are not mentioned in this list are not used or changed by the JSYS.

The programmer must initialize the input parameter accumulators and access the values returned by including appropriate assignments to and from the register data segments identified in the register name list before and after the JSYS call.

Note that it is important to minimize the scope of a specific register declaration to ensure that allocation of registers is possible.

5.26 LSH (LOGICAL SHIFT)

Calling Sequence:

```
LSH(EXP, CEXP)
```

Parameters:

EXP	Value to be logically shifted
CEXP	Number of bit positions to be shifted

Return Value:

EXP is logically shifted by CEXP bit positions. If CEXP is nonnegative, shift is to the left; otherwise, shift is to the right. Bits shifted out of EXP are replaced with zero bits.

5.27 MACHOP AND MACHSKIP (EMIT AN INSTRUCTION)

Calling Sequence:

```
MACHOP(OP, AC, Y, X, I)
MACHSKIP(OP, AC, Y, X, I)
```

Parameters:

OP	Low nine bits become the operation code of the instruction; must be a compile-time constant expression.
AC	Register name or an expression. If an expression, the low-order four bits becomes the accumulator (A) field of the instruction; otherwise, the register number corresponding to the register name must be a compile-time constant expression.
Y	The low-order 18 bits become the address (Y) field of the instruction
X	The low-order four bits becomes the index (X) field of the instruction
I	The low-order bit becomes the indirect (I) field of the instruction; must be a compile-time constant expression.

By default, any omitted parameter defaults to zero.

Return Value:

For MACHOP, the value returned is the contents of the register specified by AC after the execution of the instruction. OP must not specify an instruction that can skip.

For MACHSKIP, the value returned is 1 if the instruction skips, and 0 otherwise.

5.28 MULD (MULTIPLY DOUBLE OPERANDS)

Calling Sequence:

MULD (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A	Address of a double-precision floating-point value used as the multiplier
SRC2A	Address of a double-precision floating-point value used as the multiplicand
DSTA	Address where the product of operand 1 and operand 2 is stored

Return Value:

NOVALUE

5.29 MULF (MULTIPLY FLOATING OPERANDS)

Calling Sequence:

MULF (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A	Address of a single-precision floating-point value used as the multiplier
SRC2A	Address of a single-precision floating-point value used as the multiplicand
DSTA	Address where the product of operand 1 and operand 2 is stored

Return Value:

NOVALUE

5.30 MULG (MULTIPLY FLOAT-G OPERANDS)

Calling Sequence:

MULG (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A	Address of an extended double-precision floating-point value used as the multiplier
SRC2A	Address of an extended double-precision floating-point value used as the multiplicand
DSTA	Address where the product of operand 1 and operand 2 is stored

Return Value:

NOVALUE

5.31 POINT (BUILD A BYTE POINTER)

Calling Sequence:

```
POINT(Y, P, S, X, I)
```

Parameters:

Y	The low-order 18 bits of Y become the address (Y) field of the byte pointer. There is no default.
P	The low-order six bits of P become the position (P) field of the byte pointer. The default is 0.
S	The low-order six bits become the size field (S) of the byte pointer. The default is 36.
X	The low-order four bits become the index (X) field of the byte pointer. The default is 0.
I	The low-order bit becomes the indirect (I) field of the byte pointer. The default is 0.

Return Value:

If Y is a link-time constant expression, and P, S, X, and I are compile-time constant expressions, the return value is a link-time constant expression.

5.32 REPLACEI AND REPLACEN (STORE A BYTE)

Calling Sequence:

```
REPLACEI(AP, EXP)
REPLACEN(AP, EXP)
```

Parameters:

AP	Address of a byte pointer
EXP	Value to be stored

Return Value:

REPLACEI increments the byte pointer at address AP. Both REPLACEN and REPLACEI then store the value EXP in the field defined by the byte pointer. The value returned is EXP.

5.33 ROT (ROTATE A VALUE)

Calling Sequence:

```
ROT(EXP, CEXP)
```

Parameters:

EXP	Value to be rotated
CEXP	Number of bits positions to be rotated

Return Value:

EXP rotated by CEXP bit positions. IF CEXP is nonnegative, rotate left; otherwise, rotate right.

5.34 SCANN AND SCANI (FETCH A BYTE)

Calling Sequence:

```
SCANN(AP)
SCANI(AP)
```

Parameters:

AP	Address of a byte pointer

Return Value:

SCANI increments the byte pointer at address AP. Both SCANN and SCANI then fetch the field defined by the byte pointer and use that as the returned value.

5.35 SUBD (SUBTRACT DOUBLE OPERANDS)

Calling Sequence:

```
SUBD (SRC1A, SRC2A, DSTA)
```

Parameters:

SRC1A	Address of a double-precision floating-point value used as the subtrahend
SRC2A	Address of a double-precision floating-point value used as the minuend
DSTA	Address where the difference between operand 1 and operand 2 is stored

Return Value:

NOVALUE

5.36 SUBF (SUBTRACT FLOATING OPERANDS)

Calling Sequence:

```
SUBF (SRC1A, SRC2A, DSTA)
```

Parameters:

SRC1A	Address of a single-precision floating-point value used as the subtrahend
SRC2A	Address of a single-precision floating-point value used as the minuend
DSTA	Address where the difference between operand 1 and operand 2 is stored

Return Value:

NOVALUE

5.37 SUBG (SUBTRACT FLOAT-G OPERANDS)

Calling Sequence:

SUBG (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A	Address of an extended double-precision floating-point value used as the subtrahend
SRC2A	Address of an extended double-precision floating-point value used as the minuend
DSTA	Address where the difference between operand 1 and operand 2 is stored

Return Value:

NOVALUE

5.38 UUO (INVOKE A TOPS-10 SYSTEM SERVICE)

Calling Sequence:

UUO (skips, opcode, acexp, effexp)

Parameters:

skips	A compile-time constant expression whose value is in the range 0 to 1. The value of this expression indicates the manner in which control may return following the UUO instruction.
opcode	The low-order nine bits of this value become the operation code of the generated instruction. (Must not be a register name nor a %REF function.)
acexp	If this parameter is a register name, the number of the register becomes the accumulator field of the generated instruction. Otherwise, the low-order four bits of this value become the accumulator field of the generated instruction (must not be a %REF function).
effexp	If this parameter is a register name, the number of the register becomes the effective address of the generated instruction. Otherwise, the low-order 18 bits of this value become the effective address of the generated instruction (must not be a %REF function).

An instruction with the indicated operation code, accumulator field, and effective address is executed.

Note that the INIT UUO, which has inline parameters, is not supported by the UUO machine-specific function. The OPEN UUO may be used instead.

The value of the skips parameter must agree with the characteristics of the instruction or UUO which is executed. It defines the manner in which control may return following the instruction, and the value of the UUO function, according to the following table:

SKIPS Value	Interpretation
0	Control always returns to the instruction plus 1. The value of the function is zero.
1	Control returns either to the instruction plus 1 or to the instruction plus 2. The value of the function is zero if control returns to the instruction plus 1, and 1 if control returns to the instruction plus 2.

CHAPTER 6

PROGRAMMING CONSIDERATIONS

This chapter gives some practical help in writing BLISS programs. First, the usage differences between LIBRARY and REQUIRE files are considered. Then, some common BLISS programming errors are discussed.

6.1 LIBRARY AND REQUIRE USAGE DIFFERENCES

BLISS library files are used in a manner similar to that of required files; declarations that are common to more than one module are centralized in a single file 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 compilation cost of processing the source occurs every time the file is used in a compilation. 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, costs of lexical processing (including scanning, lexical-conditionals, lexical-functions, and macro expansions) and declaration parsing and checking occur only during the library precompilation.

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 the declarations into the normal symbol table unless and until the declaration is 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 may typically be improved by a factor of 4 by using library files instead of source files.

Library files and the same declarations used from source files using the REQUIRE declarations are similar. However, the differences are:

- Files invoked by REQUIRE declarations are source (text) files; files invoked by LIBRARY declarations must be special files created by the compiler in a previous library precompilation.

- Files invoked by REQUIRE declarations may 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]
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 may 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.

- Files invoked by REQUIRE declarations may have effects that are dependent on previous declarations and/or switch settings in the module being compiled. This can occur in a lexical-conditional (%IF-%THEN-%ELSE-%FI) or macro expansion that depends on the lexical functions %SWITCHES, %DECLARED, or %VARIANT, or on values of predeclared literals, such as %BPVAL. Files invoked by LIBRARY declarations do not have effects that are dependent 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 precompilation.

- Appropriately written source files can be invoked by REQUIRE declarations that exist in a module which may be compiled by any BLISS compiler; however, Library files must only be invoked by the same compiler that created the library file.

In compiling a module, identical effects are normally achieved by using either a REQUIRE declaration to invoke the source files used to create a library or by using a LIBRARY declaration to invoke the library itself. However, the differences presented above may lead to problems that are difficult to identify. Therefore, for each set of declarations, one form or the other should be used consistently.

6.2 FREQUENT BLISS CODING ERRORS

Certain user coding errors occur frequently, especially for new BLISS users, when compiling and debugging a new module. The following list may be useful as a check list when you cannot seem to find the source of a problem.

1. Some restrictions apply to these declarations.

6.2.1 Missing Dots

The most frequent error is to forget a dot. Except for the left side of an assignment, the appearance of a data segment name without a preceding fetch operator is the exception and usually a mistake. For example:

```
IF A THEN ...
```

should almost certainly be:

```
IF .A THEN ...
```

6.2.2 Valued and Nonvalued Routines

The BLISS compiler does not diagnose useless value expressions in contexts that do not use a value. For example, in the following routine:

```
ROUTINE R(A): NOVALUE =
    BEGIN
    ...
    RETURN 5;
    ...
    END;
```

the apparent return value 5 is discarded since the routine has the NOVALUE attribute.

However, in the following case:

```
ROUTINE S(B) =
    BEGIN
    ...
    RETURN;
    ...
    END;
```

a required RETURN value is missing; therefore, 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.)

6.2.3 Semicolons and Values of Blocks

It is common to think of a semicolon as a terminator of an expression; this is, however, untrue and can lead to errors. In the following example:

```
IF .A
THEN
    X=.Y;
ELSE
    X=-5;
```

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 6.2.2 above concerning valued and nonvalued routines.)

6.2.4 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. Explicit parentheses can, however, make your intent clear to the compiler (as well as to yourself and to other readers). For example, instead of the following:

```
IF .X (EQL 0 AND .Y (OR NOT .J THEN...
```

use:

```
IF ((.X (EQL 0) AND .Y) OR (NOT .J) THEN...
```

6.2.5 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
```

calls the routine at address R with a parameter of zero. However:

```
.L(0)
```

calls the routine at address L (most likely an address on the stack) and uses the returned value as the operand of the fetch. Since 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:

```
BLISS36C(.L,0)
```

6.2.6 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 of signed versus unsigned operations is important. For example, in the following:

```
BEGIN
FIELD LOW9 = [0,0,9,0];
OWN
   X : BLOCK[1] FIELD(LOW9);
IF .X[LOW9] EQL -5
THEN ...
...
END
```

the expression .X[LOW9] EQL -5 is always false because the (unsigned) value fetched from X is necessarily in the range 0 to 511.

6.2.7 Complex Macros

The BLISS macro facility has many capabilities but also has some very subtle properties. Most problems arise when features that have side-effects on the compilation are used, such as:

- 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 particularly careful when trying to use these features; indeed, you may not really need them in the first place. (Refer to Section 6.3 for examples of advanced macro usage.)

6.2.8 Missing Code

You may discover that some of your program seems to be missing from the compiled code. Check the compiled code carefully to make sure that it really is missing rather than cleverly optimized. If code is missing, most likely you have made a coding error. Many coding errors tend to result in code that can be optimized.

For example, consider the following:

```
BEGIN
FIELD LOW9 = [0,9];
OWN
   X : BLOCK[1] FIELD(LOW9);
IF .X[LOW9] EQL -5
THEN
    X= 100
...
END
```

In the above example, the value of the test expression .X[LOW9] EQL -5 is always false. (See Section 6.2.6 above concerning signed and unsigned fields.) As a result, the compiler produces no code for the following:

- The test expression, and
- The alternative, X = 100 (it can never be executed).

Consequently, the entire IF expression disappears from the compiled code. The problem is not erroneous compiler optimization, but a missing extension expression in the field declaration of LOW9.

A similar error occurs if the fact is overlooked that TRUE/FALSE is based on the value of the low bit of an expression. Thus the following fragment:

```
IF .X AND 2
THEN
    Y=0
ELSE
    Y=1
```

always assigns the value "1" to Y because the low bit of ".X AND 2" must always be zero (false).

6.2.9 Conflicting Names

Be aware of both the length and content of user-defined symbolic names. BLISS can distinguish among symbols that differ in any of the first 31 characters. However, linkers currently on the DECsystem-10/20 cannot distinguish beyond the sixth character. For this reason, any GLOBAL or EXTERNAL symbol should be unique within six characters. If a file is created for assembly outside of BLISS, all symbols appearing within the file should be unique within six characters or the /UNAMES switch should be specified. This includes symbols defined by OWN, GLOBAL, LABEL, EXTERNAL, GLOBAL ROUTINE, and EXTERNAL ROUTINE.

BLISS issues a diagnostic message when a name exceeds 31 characters and ignores any characters beyond that. Thus, to avoid confusion among names, distinguish a name within a name as soon as possible; for example, use IN_A_VALUE and IN_B_VALUE instead of IN_VALUE_A and IN_VALUE_B. BLISS also issues a diagnostic message when two unique GLOBAL names match in the first six characters; the linker also diagnoses this problem.

BLISS allows the declaration of the same symbolic name in different contexts, provided the declarations appear in distinct blocks. However, the MACRO assembler neither recognizes blocks nor permits any redefinition of symbols. If a file is created for assembly outside of BLISS, all symbols declared by ROUTINE, OWN, and GLOBAL must be unique within the entire module. If this is not true, the /UNAMES switch must be used to avoid an assembly error.

Good programming practice dictates that duplicate symbolic names should be avoided wherever possible, since they cause confusion and add to debugging and program maintenance time. If any duplication occurs, use the names for the same purpose in parallel construction.

This code is acceptable:

```
MACRO
    ADD(A, B) = A + B%
    SUB(A, B) = A - B%
```

but the following code is not recommended:

```
BEGIN
OWN
    A;
    BEGIN
    LITERAL
        A=5;
    ..
    END;
..
END
```

6.2.10 Routines Within Routines

Declaring a routine within the declaration of another routine must be avoided for the following reasons:

- Although an embedded routine can take a name identical to that of an unembedded routine, without error, you may inadvertently cause confusion and subsequent errors by using this name to refer to the outer routine.

- A degree of complexity is added to the compilation process, causing the compiler to take longer to compile your program.
- It is more difficult to follow the code when each routine does not "stand by itself."

6.2.11 Indexed Loop Coding Error

A common coding error occurs in the use of unsigned indexed-loop-expressions of the form:

```
DECRU I FROM HIGH TO 0 DO
```

This code results in an infinite loop because you are attempting to decrement through zero. Since the rules of unsigned arithmetic state that zero is the smallest integer, the .I will never be 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:

```
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 are GTRA and LSSA.

6.3 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)

6.3.1 Advantageous Use of Machine Dependencies

The following examples show how machine-independent constructs may 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 form:

```
VECTOR[ DST, .I ] = .VECTOR[ SRC, .I ];
```

and the DST or SRC expressions were routine-calls, a call would be executed every time a pass was made through the loop. Thus, the BIND to $D and $S ensures that this side-effect occurs only once.

Using String Instructions

The macro shown in the previous example is, however, inefficient for use in moving 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-11 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 with the use of 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 determines if the PDP-10 effective-address calculation can be used to completely evaluate the expression:

```
.SQQQ + (LENTH) - 1
```

Otherwise, 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.

It should be noted, however, 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 incorrectly moves one word to the destination.

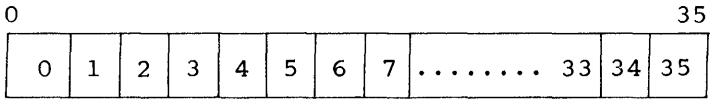
6.3.2 Dealing with Enumeration Types

One of the more powerful features of languages such as PASCAL and ADA is the ability to define a finite set of elements. This capability is known as an "enumerated type", because the set is defined by exhaustively naming each possible element in the set.

While BLISS is unaware of such types, it is possible to provide an adequate simulation.

6.3.2.1 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 domains.

A subset is stored by turning on 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 BLISS-36 a set looks as follows:



The digits inside the box refer to element-numbers, while the digits outside the box refer to the most-significant (0) and least-significant bits (35) of a word.

The following table shows some simple sets and the elements they contain:

Set (in octal)	Contents
400000000000	0
400000000001	0,35
000000000002	34
776000000000	0,1,2,3,4,5,6,7

6.3.2.2 Creating a Set - The following macro is used to define 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:

```
LITERAL
    OAK = 0,
    MAPLE = 1,
    PINE = 2,
    ELM = 3;
```

And the name TREES should be defined by the macro

```
MACRO TREES = OAK, MAPLE, PINE, ELM %;
```

The %QUOTE is necessary to prevent the %EXPAND from occurring until the ENUMERATION macro is expanding; at that time, the %REMAINING will be bound to the list of names: OAK, MAPLE, PINE, ELM.

Because the macro NAME is inside another macro, its body is being scanned at macro-quote level when the ENUMERATION macro expands. Thus, if the body of the ENUMERATION macro was defined as:

```
..
MACRO NAME = %REMAINING %QUOTE %;
..
```

The result of expanding ENUMERATION would be:

```
MACRO TREES = %REMAINING %;
```

Since TREES is defined as not having a macro-parameter list, the value of the %REMAINING would always be empty. Therefore, you need to force the expansion of the %REMAINING when the ENUMERATION macro is expanded, not when the TREES macro is expanded.

Also, if you define the ENUMERATION macro as:

```
..
MACRO NAME = %EXPAND %REMAINING %QUOTE %;
..
```

This would cause the %REMAINING to be expanded too soon (namely, 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:

```
INCR SPECIES FROM MIN(TREES) TO MAX(TREES) do ...
```

2. Using the CASE control-expression, write:

```
CASE .WOOD FROM MIN(TREES) TO MAX(TREES) OF
    SET
    [OAK]: ...
    [MAPLE]: ...
    [PINE,ELM]: ...
    [OUTRANGE]: ...
    TES;
```

3. A successor function can be defined as:

```
MACRO SUCCESSOR(T) = ( (T)+1 ) MOD MAX(TREES) %;
```

The MOD is used to cause SUCCESSOR(MAX(TREES)) to be the same as MIN(TREES).

Of course, a more general implementation would pass the SET as a parameter, as in:

```
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).

6.3.2.3 Placing Elements in Sets - Given elements of an enumeration type, you want to 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 including 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 in:

```
OWN FOREST : INITIAL( SUBSET<OAK, MAPLE> );

...
IF (.FOREST AND SUBSET<PINE,ELM>) EQL 0
THEN
    ! No trees in common between two sets
```

6.3.2.4 Membership in a Set - The ONEOF macro efficiently determines if an element is a member of a given subset. This depends on doing a left-shift and examining the sign-bit of the result value. This is why 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 leaves the sign-bit unchanged, the desired behavior 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 compiletime, BLISS-36 must generate a run-time test and conditionally execute either the ASH or LSH. Because you know that the shifts are always 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 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
   BEGIN BUILTIN LSH;
   LSH ( (1^(35-3)) OR (1^(35-2)) ), .TREE)  LSS 0
   END
THEN
```

This is equivalent to the following:

```
IF LSH (%'140000000000', .TREE) LSS 0
THEN
    ...
```

And assuming that TREE still contains its initial value of 3 (for ELM), you have:

```
IF (%0'400000000000') LSS 0
THEN
    ...
```

This evaluates to TRUE and indicates that ELM is a member of the subset {ELM, PINE}.

6.4 EXTENDED ADDRESSING DIFFERENCES

Compiling a BLISS-36 program with extended addressing differs from the compilation of programs without this feature in the following ways:

- The value of %BPADDR is 30.
- If a legal character size (6, 7, 8, 9, 18) is used, the CH$PTR function returns a 1-word global byte pointer; otherwise, a local byte pointer is returned.
- The compiler assumes that all relocatable symbols and externals are in the same section; this includes the following data- and routine-declarations:

```
FORWARD, OWN, GLOBAL, EXTERNAL,
FORWARD ROUTINE, ROUTINE,
GLOBAL ROUTINE, EXTERNAL ROUTINE
```

- The compiler performs all address arithmetic to 30 bits of precision. To accomplish this, an XMOVEI (30-bit access) instruction is generated in place of a MOVEI (18-bit access) instruction. For example:

```
EXTERNAL
    a;
LOCAL
    b;
b = a;
  .
  .
  .
```

The extended addressing code generates:

```
XMOVEI AC,A     ;B,A
```

Without extended addressing the code would be:

```
MOVEI AC,A      ;B,A
```

- Since the LINKER restricts the use of nonzero sections to PSECTED programs, PSECTED code is generated by default.

If, in addition, you use the SECTION-INDEPENDENT option to extended addressing, the following difference occurs:

- The compiler generates code that, at run time, determines the section number. For example:

```
OWN
    A,
    B;
B = CH$PTR(A);
  .
  .
  .
```

The code in the example generates the following sequence of assembly instructions:

```
SECTION_INDEPENDENT CODE:
    !
    ! The address of A-1 is determined at run-time.
    !
    XMOVEI AC, A-1
    TLO    AC, -120000
    MOVEM  AC, B
     .
     .
     .
NOSECTION_INDEPENDENT CODE:
     !
     ! The address of A-1 is determined at link-time.
     !
     MOVE   AC, C.1
     MOVEM  AC, B
     .
     .
     .
C.1: XPOINT 7, A-1, 34
```

CHAPTER 7

TRANSPORTABILITY GUIDELINES

This chapter addresses the task of writing transportable programs. It shows why writing such transportable code is much easier if considered from the beginning of the project, explores properties that cause a program to lose its transportability, and discusses techniques by which a programmer can avoid these pitfalls.

After an introduction to the concept of transportability, the transportability guidelines presented in this chapter are organized into three sections. The section on general strategies discusses some high-level approaches to writing transportable software in BLISS. The section on tools describes various features of the BLISS language that can be used in solving transportability problems. The section on Techniques analyzes various transportability problems and suggests solutions to them.

The dialects discussed in this chapter are the languages defined by the following BLISS compilers:

BLISS-16 V4
BLISS-32 V4
BLISS-36 V4

7.1 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-11. Various solutions to the problem of transportability exist, each requiring different levels of effort. Various kinds of solutions are recommended. In some cases, for example, program text should be rewritten. In other cases, large portions of programs may be written in such a way that no modification is required and equivalent functionality is preserved in differing architectures. The levels of solutions in order of decreasing desirability are:

- No change is needed to program text. 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. Some of what is contained here will not be obvious to programmers. An attempt is made to identify those areas that can cause problems, if the programmer is not forewarned. Solutions to all identified problems are suggested.

7.2 GENERAL STRATEGIES

This section presents certain global considerations that are important to the writing of transportable BLISS programs, namely:

- Isolation
- Simplicity

7.2.1 Isolation

You should keep in mind the following maxim when designing and/or coding a program that is to be transported:

- If it is nontransportable, isolate it.

You will probably encounter situations for which it is desirable to use machine-specific constructs in your BLISS program. In these cases, simply isolating the constructs will facilitate any future movement of the program to a different machine. In most cases, only a small percentage of the program or system will be sensitive to the machine on which it is running. By isolating those sections of a program or a system, 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 is to be allocated, place the allocation in a separate MODULE or in a REQUIRE file.
- If machine-specific data is to be accessed, place the accessing code in a ROUTINE or in a MACRO, and then place the ROUTINE or MACRO in a separate MODULE or in a REQUIRE file.
- If a machine-specific function or instruction is to be used, isolate it by placing it too 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 this code is nontransportable.

The above 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 which are machine or operating system dependent from the rest of the system, you can simplify the task of transporting the entire system. It becomes a matter of recoding 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 recoding effort. BLISS is a language which facilitates both the design and programming of programs and systems in a modular fashion. This feature should be used to advantage when writing a transportable system.

7.2.2 Simplicity

A basic concept in writing transportable BLISS software is simplicity in the use of the language. BLISS was originally developed for implementing systems software. As a result of this, BLISS is nearly unique among high-level programming languages in that it allows ready access to the machine on which the program will be running. The programmer is 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 inherently 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, coding transportable programs is a simpler task because the number of options available has been reduced. Simplicity in the coding effort is one of the reasons for the development of higher-level languages like BLISS. Using the defaults in BLISS will result in programs which are much more easily transported.

7.3 TOOLS

This section on tools presents various language features that provide a means for writing transportable programs. These features are either intrinsic to BLISS or have been specifically designed for transportability and/or software engineering uses. The tools described here will be used throughout the companion section on techniques.

7.3.1 Literals

Literals provide a means for associating a name with a compile-time constant expression. This section considers some built-in literals which will aid in writing transportable programs. In addition, it discusses restrictions on user-defined literals.

7.3.1.1 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 most 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:

Description	Literal Name	10/20	VAX-11	11
Bits per addressable unit	%BPUNIT	36	8	8
Bits per address value	%BPADDR	18	32	16
Bits per BLISS value	%BPVAL	36	32	16
Units per BLISS value	%UPVAL	1	4	2

The names beginning with '%' are literal names that can be used without declaration. These literal names are used throughout these guidelines.

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.

We can derive other useful values from these built-in literals. For example:

```
LITERAL
    HALF_VALUE = %BPVAL / 2;
```

defines the number of bits in half a word (half a longword on VAX-11).

7.3.1.2 User-Defined Literals - A literal is not strictly speaking a self-defining term. The value and restrictions associated with a literal are arrived at 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/or 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-11, the size is 32 bits; on the 10/20 systems, it is 36; and the 11 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:

```
VAX-11:  -(2**31)          ≤ i ≤ (2**31) - 1

 10/20:  -(2**35)          ≤ i ≤ (2**35) - 1

    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 value is the ASCII code corresponding to the character in quotes and when stored, it is right-justified in a BLISS value (word or longword). A more thorough discussion of its usage can be found in the section on character sequences.

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 a %ASCII type string literal:

	VAX-11	10/20	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 the section Character Sequences.

7.3.2 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 on structures. There, 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. These macros can be used to selectively compile certain declarations and/or expressions depending on which compiler is being run. There are three sets of definitions, each containing three macro definitions.

The definitions for the BLISS-32 set are:

```
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
```

7.3.3 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 and
- 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:

```
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 will assume 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 will give 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 for.

NOTE

The precise set of language constructs that are subject to transportability checking is specified in Appendix C of the Bliss Language Guide.

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 the BLISS-16 and BLISS-32
        target architectures.)

        The compilation of this section
        of code by a BLISS-36 compiler will
        result in a diagnostic warning.

                ...
                ...
                ...

END;

!+
!  Full transportability checking is restored.
!-
```

7.3.4 Reserved Names

The following is a list of the BLISS reserved names. These names cannot be user declared. Note that, while 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 Appendix A of the Bliss Language Guide for a more complete description.

*ADDRESSING_MODE
*ALIGN
ALWAYS
AND
BEGIN
BIND
*BIT
BUILTIN
BY
*BYTE
CASE
CODECOMMENT
COMPILETIME
DECR
DECRA
DECRU
DO
ELSE
ELUDOM
ENABLE
END
EQL
EQLA
EQLU
EQV
EXITLOOP
EXTERNAL
FIELD
FORWARD
FROM
GEQ
GEQA
GEQU
GLOBAL
GTR
GTRA
GTRU
IF
INCR
INCRA
INCRU
INITIAL
INRANGE
*IOPAGE
KEYWORDMACRO
LABEL
LEAVE
LEQ
LEQA
LEQU
LIBRARY
LINKAGE
LITERAL
LOCAL
*LONG
LSS
LSSA
LSSU
MACRO
MAP
MOD
MODULE
NEQ
NEQA
NEQU
NOT
NOVALUE
OF
OR
OTHERWISE
OUTRANGE
OWN
PLIT
PRESET
PSECT
*RECORD
REF
REGISTER
REP
REQUIRE
RETURN
ROUTINE
SELECT
SELECTA
SELECTONE
SELECTONEA
SELECTONEU
SELECTU
SET
*SHOW
*SIGNED
STACKLOCAL
STRUCTURE
SWITCHES
TES
THEN
TO
UNDECLARE
*UNSIGNED
UNTIL
UPLIT
VOLATILE
*WEAK
WHILE
WITH
*WORD
XOR

7.3.5 Require and Library Files

REQUIRE files are a way of gathering machine-specific declarations and/or 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 code a particular BLISS construct (for example, routines, data declarations, etc.) 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 are very machine specific, then the solution may be to code 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:

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:

```
REQUIRE
    'IOPACK';           ! I/O Package
```

will search for IOPACK.R36, IOPACK.R16 or IOPACK.R32, depending on which compiler is being run. Failing that, it will look 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 using the /LIBRARY switch, is capable of precompiling files containing declarations. Not all the declarations can be processed in a library run however; those that are allowed are described elsewhere. 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 which is to run on a DECSYSTEM-20 and a VAX. To precompile it requires that we run the BLISS-32 compiler on it using 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).

7.3.6 Routines

The key to transportability is the ability to identify an abstraction that can exist in several environments. This is done by naming the abstraction and describing its external characteristics in a way that

permits implementation in any of the environments. The abstraction may 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, like structures, macros, literals, require files, etc., 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, there will be a 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 when writing cross-compilers, for instance. Even if the ability to perform cross-arithmetic is not desired, isolating floating point operations in routines may be a good idea since 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 (since 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 nontrivial, and that cost is strictly program overhead. Composing this sort of abstraction in the limit will produce serious performance degradation. For this reason, a programmer should probably try not to use the routine as a transportability mechanism if a small amount of forethought will be sufficient to enable the writing of a single transportable module.

7.4 TECHNIQUES

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:

- 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.

7.4.1 Data

This section deals with the allocation of data in a BLISS program. In this section we do not deal with character sequence (string) data or the formation of address data. These types of data are discussed in their own sections (see "Data: Addresses and Address Calculation" and "Data: Character Sequences"). Primarily, we discuss the allocation of scalar data (for example, counters, integers, pointers, addresses, etc.) A presentation of more complex forms of data can be found in the

sections entitled "Structures and Field-Selectors" and "PLITs and Initialization." First there is a discussion of transportability problems encountered due to differing machine architectures. Next a discussion of the BLISS allocation-unit attribute is presented. Finally, a discussion of other BLISS data attributes that must be considered when writing transportable programs is discussed.

7.4.1.1 Problem Origin - The allocation of data (via the OWN, LOCAL, GLOBAL, etc. 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:

- A machine's architecture
- The flexibility of the BLISS language

When we are considering writing a BLISS program that will be transported to another machine, we are confronted with the problem of allocating data on (at least two) architecturally different machines.

Although we have already discussed differing word sizes, there are further differences. On the VAX-11 architecture, data may be typically fetched in longwords (32 bits), in words (16 bits) and in bytes (8 bits); on the 11, both words and bytes may be fetched. Only 18-bit halfwords and 36-bit words on the 10/20 systems may be fetched without a byte pointer.

If we were writing our program in an assembly language we would not consider these differences to be important; clearly, our assembly language program was not intended to be transportable.

What decisions, however, must the BLISS programmer make in the transportable allocation of data? Need he or she be concerned with how many bits are going to be allocated?

These questions (and their answers) can be 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 is fortunate for the programmer who is writing highly machine-specific software for efficiency purposes. This programmer needs much more control over exactly how many bits of data will be used. This feature of BLISS, however, can complicate the decisions that need to be made by the BLISS programmer who is writing a transportable program. Does he or she allocate scalars by bytes, or by words, or by longwords?

7.4.1.2 Transportable Declarations - Consider the following simple example of a data declaration in BLISS-32:

```
LOCAL
    PAGE_COUNTER: BYTE;          ! Page counter
```

The programmer has allocated one byte (8 bits) for a variable named PAGE_COUNTER. No matter what the intentions were in requesting only one byte of storage, this declaration is nontransportable. The concept of BYTE (in this context) does not exist on the 10/20 systems.

In fact, in BLISS-36 the use of the word BYTE results in an error message. Since this storage is allocated on the stack or in a register, there is even less motivation to make it a byte due to the high reusability of these locations.

If this declaration had been originally coded as:

```
LOCAL
    PAGE_COUNTER;                   ! Page counter
```

then 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. We allowed the BLISS compiler to allocate storage by default by not specifying any allocation-unit in the LOCAL declaration. In all the BLISS dialects the default size for allocation-unit consists of %BPVAL bits. Thus our first transportable guideline is:

- Do not use the allocation-unit attribute in a scalar data declaration.

The use of the default allocation-unit will sometimes result in the allocation of more storage than is strictly necessary. This gain in program data size (which, in most instances, is small) should be weighed against a decrease in fetching time for a particular scalar value, 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, most efficiently accessed unit of data for a particular machine. In other words, the saving of bits does not necessarily mean a more efficient program.

Besides the allocation-unit there are other attributes that may present transportability problems if used. In particular, when allocating data:

- Do not use the following attributes:

  Extension (SIGNED and UNSIGNED),
  Alignment,
  Weak

  which is to say: think twice before you write a declaration. Do you really need to specify any data attributes other than structure attributes?

The extension-attribute specifies whether the sign bit is to be extended in a fetch of a scalar (or equivalently, whether or not the left most 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 tells the compiler at what kind of address boundary 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 is a VAX/VMS-specific attribute and is not supported by BLISS-36 or BLISS-16. It therefore cannot be used in a transportable program.

These guidelines are relatively simple, yet they should relieve the BLISS programmer of needing to worry about how the program data will actually be allocated by the compiler. 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 the use of one or more of the above attributes. In fact, it may be desirable to take advantage of a specific machine feature. In these cases follow this guideline:

- Conditionalize and/or heavily comment the use of declarations which may be nontransportable.

This guideline is the "escape-hatch" in this set of guidelines. It should only be used sparingly and where justified. To use it often will only result in more code that will need to be rewritten when the program has to be transported to another machine - and rewriting code is not a goal.

7.4.1.3 Length of Externally Used Names - The length of names declared in EXTERNAL and GLOBAL declarations should be no longer than six characters. Linkers and task builders on PDP-11, DECsystem-10, and DECSYSTEM-20 systems can not distinguish between symbols in which the first six characters are identical.

7.4.2 Data: Addresses and Address Calculations

This section discusses address values and calculations using address values. First, there is a presentation of problems that might occur when an address or the result of an address calculation is used as a value. A transportable solution to some of these problems is then presented. Next, a discussion is presented on the need for address forms of the BLISS relational operators and control expressions and how and when to use them. Finally, some important differences in the interpretation of address values between BLISS-10 and BLISS-36 are discussed.

7.4.2.1 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. 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 the section on "Literals."

To see how these size differences can have an effect on writing transportable programs, consider a common type of address expression; namely an expression that computes an address value from a base (a pointer or an address) and an offset. That is, some expression of the form:

```
... 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 (most likely) was 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. The effect, however, is different on each machine as will be shown.

By adding 1 to an address (in this case, INPUT_RECORD) the address of the next addressable unit on the machine is being computed. In BLISS-32 and BLISS-16 this would be the address of the next byte (8 bits), but in BLISS-36 this would be the address of the next word (36 bits). This is probably not a transportable expression because of the different sizes of the addressable units and the resultant values.

Based on the above example, follow this 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 already exists as another predeclared literal. Its definition is %BPVAL/%BPUNIT, and it is called %UPVAL.

Using this literal in our example yields:

```
ELEMENT_2 =
   .(INPUT_RECORD + 1 * %UPVAL);
```

The address expression is now tranportable.

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 coded as:

```
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 the section on structures and field selectors.

7.4.2.2 Relational Operators and Control Expressions - The previous example illustrated the use of address values in the context of computations. Other common uses of addresses are in comparisons (testing for equality, etc.) and as indices in loop and select expressions. The use of 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.

It can be seen that to perform a simple relational test of address values:

```
... ADDRESS_1 LSS ADDRESS_2 ...
```

requires two different interpretations. This expression would evaluate correctly on the 10/20 systems. But, on VAX-11 and 11 machines, the following would have had to have been coded for the comparison to have been made correctly:

```
... 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, etc.) which support address comparisons.

In BLISS-16 and BLISS-32, the address-relationals are equivalent to the unsigned-relationals. In BLISS-36, the address-relationals are equivalent to the signed-relationals. For all practical cases, a user need not be concerned with this, since this "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 contol 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.

A violation of either guideline causes unpredictable results.

7.4.2.3 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. However, BLISS-36 defines the value simply as the address in the low half and zeros in the high half. This change was made solely for reasons of transportability, since 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. Thus, the expressions:

```
Y = .X;
Y = F(.Y) + 2;
```

are the same in both BLISS-10 and BLISS-36, but

```
Y = X;
```

assigns a different value to Y in BLISS-36 and in BLISS-10.

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. Thus the meaning of:

```
Y<0,18> = .X<3,7>;
```

is unchanged, but

```
Y = X<3,7>;
```

is invalid. Moreover, it is highly recommended that field selectors never appear outside of a structure declaration, since position and size are apt to be highly machine dependent. A thorough discussion can be found in the section on structures and field selectors.

7.4.3 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 were the rule, having been implemented by individuals or projects to suit their particular needs. This section views quoted strings in two contexts: as values (integers) and as character strings. Transportability problems associated with quoted strings and guidelines for their use are discussed.

7.4.3.1 Usage as Numeric Values - The use of quoted strings as values (in assignments and comparisons) illustrates the problem of differing representations on differing architectures. Describing the natural translation of a string literal for each architecture will illustrate the problem. 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 10/20 systems 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 10/20 string-literal had been right-justified in the word, it still would not equal the VAX-11 representation, numerically. So, in fact, the following would not be transportable:

```
WRITE_INTEGER( 'ABC' );
```

since '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 our first guideline is:

- 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:

```
%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.

7.4.3.2 Usage 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 very complete and powerful set of operations on characters. The next guideline is:

- Use the built-in character handling package when allocating and operating upon character sequences. This is the only way one can guarantee the portability of strings and string operations.

A more detailed description of these functions can be found in the Character Handling Functions chapter of the Bliss Language Guide.

7.4.4 PLITs and Initialization

This section is primarily concerned with PLITs and their uses. 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 will be illustrated by the development of a transportable table of values.

7.4.4.1 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 (or UPLIT). Hence, PLITs are a means for defining references to multi-word constants. PLITs are often used to initialize data segments (for example, 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 may be strings, numeric constants, address constants, or any combination of them, provided the value of each is known prior to execution time.

7.4.4.2 Scalar PLIT Items - The first transportability problem that might be encountered with the use of PLITs is in the specification of scalar PLIT items. As with any other declaration of scalar items (pointers, integers, addresses, etc.) it is possible to define them with an allocation-unit attribute. For example, in BLISS-32, machine specific sizes 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.

Thus, the BIND should have been coded as follows:

```
BIND
    Q1 = PLIT(1, 2, 3, -4);
```

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 the section on data. Further guidelines are presented in the section on initializing packed data.

7.4.4.3 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 the section on character sequences.

Care must be exercised when strings are to be used 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. If it is specified as:

```
BIND
    CONABC = PLIT('ABCDE', ABC_ROUT);
```

then the VAX-11 representation is as follows:

```
CONABC:                 / D C B A /  (32)
                        /       E /  (32)
                        / address /  (32)
```

The representation on the 11 is:

```
CONABC:                     / B A /  (16)
                            / D C /  (16)
                            /   E /  (16)
                          / address /  (16)
```

The 10/20 representation is:

```
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 expression:

```
... CONABC + 1 ...
```

in the BLISS-36 version, but not in the BLISS-32 or BLISS-16 versions. For the BLISS-32 version, the access expression is:

```
... CONABC + 8 ...
```

and for BLISS-16, it would have to be:

```
... 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 the section on addresses and address calculations.

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, one would write:

```
... CONABC + 1*%UPVAL ...
```

But this will not work for the VAX-11 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 the use of 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 coded:

```
BIND
    CONABC = PLIT(ABC_ROUT, 'ABCDE');
```

and this expression:

```
... CONABC + 1*%UPVAL ...
```

would have resulted in the address of the second element in the PLIT (in this case the string).

7.4.4.4 An Example of Initialization - As mentioned in the beginning of this section, PLITs are often used to initialize data segments such as tables. A data segment allocated by an OWN or GLOBAL declaration can be initialized by using the INITIAL attribute. The INITIAL attribute specifies the initial values and consists of a list of PLIT items.

A good example which shows how relatively easy it is to initialize data in a transportable way is to illustrate the process one might use to build a table of employee data. Information on each employee will consist of three elements: an employee number, a cost center number and the employee's name. The employee's name will be a fixed-length, 5-character field.

For example, a line of the table would contain the following information:

```
345     201         SMITH
```

Converting this line into a list of PLIT items that conform to this section's guidelines would result in the following:

```
(345, 201, 'SMITH')
```

Notice that no allocation units were specified and that the character string was specified last. This line will now be used to initialize a small table of only one line. The table will have the built-in BLOCKVECTOR structure attribute. The table declaration would look like:

```
OWN
    TABLE:
        BLOCKVECTOR[1,3]
        INITIAL(
                345,
                201,
                'SMITH'
                );
```

However, a problem has developed. This definition would work well in BLISS-36. That is, three words would have been allocated for TABLE. The first word would have been initialized with the employee number; the second word with the cost center; and the third with the name. However, the declaration would be incorrect in BLISS-32 or BLISS-16, simply because not enough storage would have been allocated for all the initial values. BLISS-32 would have required four longwords, and BLISS-16, five words.

The problem arises as a result of the way in which strings are represented and allocated on the three machines. The solution is simple. We only need to determine the number of BLISS values that will be needed for the character string on each machine. There is a function that will give this value. It is named CH$ALLOCATION and it is part of the character handling package. It 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.

We 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 we can be assured that enough words will be allocated on each machine. No recoding will be necessary.

We 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. We will not develop it step by step, but point out and explain some of the highlights. The example:

```
                ...
                ...
                ...

!+
!       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, we 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). What if, however, we wished to have the table hold variable length names? That is, for certain reasons, we wished 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 we were to use variable length character strings, either too much or not enough storage would be allocated. And there would be no consistent way of accessing the employee name (where would the next one start?). We could, if we knew the length of every employee name, determine in advance the number of words needed. But this is not a very 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, which will construct 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')

                );
        ...
        ...
        ...
```

7.4.4.5 Initializing Packed Data - In this section we will discuss some transportability considerations involved in the initialization of packed data. By packed data, we mean 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 would be represented by the following expression:

```
v1^p1 OR v2^p2 OR ... OR vn^pn
```

where

```
max(p1, p2, ..., pn) ≤ %BPVAL

s1 + s2 + ... + sn ≤ %BPVAL
```

and for all i

```
-2**si ≤ vi ≤ 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. Notice 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 that was developed above. When we access a field within a block, 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:

```
FIELD EMP =
    SET
    EMP_ID = [0,0,%BPVAL,0],
    EMP_COST_CEN = [1,0,%BPVAL,0],
    EMP_NAME_PTR = [2,0,%BPVAL,0];
    TES;
```

These field names can be used in developing an initialization macro, by using parametric values. This is another example of how parameterization can be used 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, we 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, since we know that both DECsystem-10 and DECSYSTEM-20 hardware have instructions that operate efficiently on half-words.

If the interest is only in transporting BLISS-36 and BLISS-32, the field declaration would look like:

```
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, a macro can be designed 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'

                ));
```

What has been illustrated in the previous example is the parameterization of certain values such as field sizes. In transporting this program, benefits can be derived from the localization of 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 purposes. 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 under "FIELD Declarations" and "BLOCK Structures" in the Bliss Language Guide. Here, we are concerned only with initializing this type of structure.)

The DCB BLOCK consists of five fields. Four of the 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, we do not have to modify the design of the BLOCK in any way. Typically, however, one would design the structure for each target machine. One method of doing this is by placing its definition in a REQUIRE file. We prefer, however, to again use the technique of parameterization. We will again make use of the field reference macros as we did in the previous example.

Here is the example describing a method in which it could be initialized. The structure has been extended by making it a BLOCKVECTOR. The example:

```
!+
!  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
            ...
```

Note that this structure could be transported 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.

7.4.5 Structures and Field Selectors

Two BLISS constructs will be discussed in this section: structures and field selectors. While the use of one does not necessarily imply the use of the other, we will see that for transportability reasons field selector usage will be confined to structure declarations. Hence, these two constructs need to be discussed together.

We will begin with a general discussion of structures, in which it will be shown that a certain machine specific feature of structures can be used in a transportable manner. The best way to illustrate the process of writing transportable structures is to take the reader through the intellectual considerations that contribute to its design, so the development of a transportable structure - FLEX_VECTOR - will be presented. At this point field selectors will be discussed. Finally, a more general structure, GEN_VECTOR, will be developed.

7.4.5.1 Structures - Structure declarations are sensitive to transportability in that one may 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 which 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) the number of bits needed to hold an address (%BPADDR) and the number of bits occupied by the smallest addressable unit (%BPUNIT).

7.4.5.2 FLEX_VECTOR - An application of this is illustrated by developing 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, we want to be able to specify the vector element size will be specified in terms of the number of bits. It should be noted that 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 9 bits, or at least 9 bits. For this example, we choose the smallest natural unit whose size is greater than or equal to 9 bits. Since there are no 9-bit conveniently addressable units on any of the machines, we have a choice of 8, 16, 32 or 36-bit units.

We can see that 9 bits will fit in the only addressable unit on the 10/20 systems - the word. On the 11 we will need two bytes or a 16-bit word and on the VAX-11 we will again need two bytes.

How then can a structure be developed that will do this allocation and will also be transportable and usable on the three systems? Clearly the structure will need some knowledge of the machine architecture. This is where the role of parameterization comes in.

The predeclared literals have all the information we 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 parmeter (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:

```
STRUCTURE
    FLEX_VECTOR[ I; N, UNIT = %BPUNIT, EXT = 1 ] =
```

Notice that by setting UNIT equal to %BPUNIT the default (if UNIT is not specified) will be %BPUNIT.

The next step is to develop the formula for the structure-size expression. The expression will make use of the allocation formals UNIT and N, and in addition, the value of %BPUNIT.

If UNIT were only allowed to assume values of integer multiples of %BPUNIT (that is, 1*%BPUNIT, 2*%BPUNIT, etc.), 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 %BPUNIT's to hold the entire vector. The reader should try some values of UNIT for differing %BPUNIT in order to see how this expression evaluates.

This subexpression:

```
(UNIT + %BPUNIT - 1) / %BPUNIT
```

which we will call NO_OF_UNITS is very important in effecting the transportability and flexibility of this particular structure. The key to transporting this structure is the knowledge that it has of the value of a certain machine architectural parameter: bits per addressable-unit. This particular expression makes use of this knowledge, hence, it can adapt to any machine. This subexpression will be used twice more in the structure-body expression.

The structure-body is an address-expression. This expression will consist of the name of the structure (the base address) plus an offset based on the index I. In addition, a field selector will be needed to access the proper number of bits at the calculated address.

The offset is simply the expression NO_OF_UNITS multiplied by the index I. (Remember that indices start at 0). 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 will look like:

```
(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:

VAX-11

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>

11

UNIT = 0 to 16	same as VAX-11

10/20

UNIT = 0	no storage (FLEX_VECTOR)<0,0,1>
UNIT = 1 to 36	[[N]] Words (FLEX_VECTOR + I)<0,36,1>

The above table illustrates that if the default value for UNIT were set to %BPVAL, this structure would be equivalent to a VECTOR of longwords on VAX-11, and a VECTOR of words on the 10/20 and 11 systems.

Elements in a data segment which have 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, then only change needed would be the size parameter of the field selector. This expression then becomes:

```
... 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-11 and 10/20 machines and a series of masks and shifts for the 11.

7.4.5.3 Field Selectors - In the last structure declaration, it was necessary to make use of a field selector. Now, the use of field selectors in a more general context will be discussed.

The use of field selectors can be nontransportable because they make use of the value of the machine word size. The unrestricted usage of field selectors may 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:

Machine:

10/20	11	VAX-11
$0 \leq p$	$0 \leq p$	
$p + s \leq 36$	$p + s \leq 16$	
$0 \leq s \leq 36$	$0 \leq s \leq 16$	$0 \leq s \leq 32$

From the table we can see that:

- The most restrictive is the 11.
- The moderate restrictions are those of the 10/20.
- The least restrictive is VAX-11.

In order to ensure the transportable use of field selectors, we 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 may 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 may easily be placed in REQUIRE files.
- Complex coding making heavy use of the predeclared literals is limited to declarations.

Another transportable structure will be developed which will be affected by the table of field selector value restrictions.

7.4.5.4 GEN_VECTOR - The reader has probably noticed 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 7 bits on the 11 and VAX-11 to 27 on the 10/20 systems.

A variation of FLEX_VECTOR can be developed which will 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 10/20 word and three into a VAX-11 longword.

This structure, which will be named GEN_VECTOR, will pack as many elements as possible into a BLISS value and so will make use of the machine specific literal %BPVAL. But, since allocation is in terms of %BPUNIT, a literal will be 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 will not cross word boundaries. This constraint is because of the restrictions placed on the value of the position parameter of a 10/20 and 11 field selector. For the same reason elements cannot be longer than %BPVAL, as given in the table of field selector restrictions above.

As in FLEX_VECTOR, the allocation expression of GEN_VECTOR will need to calculate the number of allocation units needed by the entire vector. This will again be based on the number of elements (N) and the size of each element (S). But because the elements will be packed, the expression will be slightly more complicated.

The first value is the number of elements that will fit in a BLISS value. The expression:

```
(%BPVAL/S)
```

will compute this value. 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 is not allocated by words on both of the machines. Multiplying this value by %UPVAL will result in the number of allocation units needed by the vector:

```
((N/(%BPVAL/S))*%UPVAL)
```

For clarity's sake and because this expression will be used again, it will be 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, in fact, N were an integral multiple of the number of elements in a word then this macro would be sufficient for allocation purposes.

Since 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 will calculate 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 a "fudge 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:

```
[WHOLE_VAL(N,S) +
(PART_VAL(N,S) + %BPUNIT - 1)/%BPUNIT]
```

As it works out, the structure-body expression for GEN_VECTOR will be 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. This offset is some number of addressable units and is a function of

the value of the index I. The expression which will calculate this number of addressable units is the macro WHOLE_VAL. Thus, the first part of the accessing expression is:

```
GEN_VECTOR + WHOLE_VAL(I,S)
```

Note that the macro was called with the index parameter I.

This expression will result in the structure being aligned on some addressable boundary. But since the element may not begin at this point (that is, the element may 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 will calculate 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 run time, based on the value of the index I.

This completes the definition of GEN_VECTOR. The entire declaration is:

```
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>;
```

The reader should compile this structure and see how it works in BLISS-16, BLISS-32 and BLISS-36.

7.4.5.5 Summary - 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 or BLOCKVECTORS have not been discussed, but it is hoped that the reader will get from the examples a feeling for the techniques involved in transporting structures.

There is no easy solution to transporting data structures. One should consider, when developing data structures, the machines that the program or system is targeted for and make full use of the predeclared literals such as %BPUNIT.

This exercise in the development of transportable structures has illustrated 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.

The accessing of odd (not addressable) units of data is accomplished by the use of field selectors. The field selector should only be used in structure declarations.

CHAPTER 8

COMPILER OVERVIEW AND OPTIMIZATION SWITCHES

This chapter provides an overview of the BLISS compiler organization and processing. The material presented here assumes that the reader has 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 connection with "SWITCHES declaration" in the Bliss Language Guide 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:

CODE and NOCODE
OPTIMIZE and NOOPTIMIZE
OPTLEVEL
SAFE and NOSAFE
ZIP and NOZIP

Table 1-1 shows command qualifier relationships to these switches.

8.1 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 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. This level of detail is not important in the following discussion of the phases. Simply note that the term "phase" should not be taken literally.

8.1.1 Lexical and Syntactic Analysis

The lexical and syntactic analysis 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 1-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 a "syntax-only" mode, in which the LEXSYN phase does not produce the tree representations for expressions and the later phases are not performed; moreover, if an error (as contrasted with 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. This is due to the fact that some errors are only detected and reported by later phases of the compiler and these phases are not performed.

The difference between an error and a warning diagnostic is based on the seriousness of the effect of the error upon the internal representation of the program used by the compiler. (It is not a value judgment upon the nature of the programmer's mistake.)

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.

8.1.2 Flow Analysis

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 that will 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 may 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 may 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, which might be more efficient in time or space or both. Note that the next phase 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 totally 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 the effects described below.

To understand the effects of the OPTIMIZE and SAFE switches, it is first necessary to understand more about how flow analysis is performed.

8.1.2.1 Knowing When a Value Changes - One operator in BLISS, the assignment operator, can change the contents of a data segment. However, routine calls can also 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 same analysis is performed 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 using examples. In these examples the following 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 possible common subexpression .X[.I] is not recognized by the compiler. (However, note that the common subexpression X[.I], which computes the address of the i'th element of X, is recognized since 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 virtually 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 and that cannot be referenced in the called routine by the name used to allocate the storage

The NOSAFE switch may be used to override the default assumption described above. (SAFE is the default). If NOSAFE is specified, the compiler assumes that indirect assignments do change some named data segment. Because it is nearly always impossible to identify the data segment that is changed, this assumption is guaranteed by making the even stronger assumption that all named data segments are changed.

8.1.2.2 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, the expression

```
F(X) + .X
```

may 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. Since the routine call F(X) may change the contents of X, the question becomes: When does the compiler take the (potential) change into effect? It does not make sense to take this into account within the expression without also specifying precisely 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 "!" is used to point to the mark point within the language syntax on the subsequent line.

Mark Points

```
          !
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, consider the following:

```
BEGIN
Y = .X+2;
Z = .X+2+F(X);
W = .X+2
END
```

In the second line, the content of Y is changed. This change is taken into account by the compiler 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, since common subexpressions that occur between markpoints are still detected. For example, in the expression

```
Y = (.X*2)+F(.X*2)
```

the subexpression .X*2 is computed once and the resulting value used twice, even when NOOPTIMIZE is specified.

8.1.3 Heuristics

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 subexpressions that occur only once (that is, are not common subexpressions) that should be computed in the same temporary location (whether a register or memory), thereby maximizing the use of 2-operand (as contrasted with 3-operand) instructions.

None of the switches affect the operation of the DELAY phase.

8.1.4 Temporary Name Binding

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:

- Determines the lifetime (that is, 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 loops tends to execute faster, even though the overall size of the routine may become larger.

8.1.5 Code Generation

The code generation phase, CODE, processes the tree and generates instructions. Since 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.

The /DEBUG qualifier modifies code generation as follows:

- A frame pointer (FP) is materialized and the caller's FP is saved on the stack.

- Routine parameters are popped from the stack after the routine call.

- Routine entries and exits are marked with a DEBUG UUO.

8.1.6 Code Stream Optimization

The code stream optimization phase, FINAL, processes the code stream produced by CODE and makes further optimizations at the machine code level. The optimizations performed include:

- Peephole optimization: One sequence of instructions is replaced with an equivalent shorter sequence.

- Cross-jumping: Identical sequences of code that flow into a common instruction are merged into a single sequence.

The OPTLEVEL switch may be used 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.

8.1.7 Output File Production

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.

8.2 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 specifies do not optimize across mark points.
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, non-adjacent peephole substitutions 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 either 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, etc.) to override OPTLEVEL at any time. In a multi-module compilation, each module begins with the setting defined in the command line and OPTLEVEL=2 if OPTLEVEL has not been specified in the command line.

Optimizations performed at each setting of the OPTLEVEL switch are:

	Optimization	OPTLEVEL			
		0	1	2	3
* 1.	Common subexpression detection			X	X
* 2.	Code motion out of loops, etc.			X	X
3.	Targetting/preferencing to temporaries	X	X	X	+
4.	Cross-jumping			X	X
5.	Multiple POPJs (instead of only one return point)		X	X	X
6.	Peepholes	-	X	X	X
* 7.	"SAFE" optimizations			X	X
* 8.	"OPTIMIZE" over mark points (for example, ;)			X	X
* 9.	"ZIP" speed/space tradeoff (faster but possibly larger)				X

Key

* - Another switch can control this optimization separately
X - Allowable optimization
+ - Allowed -- with increased freedom
- - Allowed -- with certain restrictions

CHAPTER 9

TOOLS, LIBRARIES, AND SYSTEM INTERFACES

A number of programming tools, libraries, and system interfaces are distributed with the BLISS-36 compiler for use by BLISS programmers. This chapter briefly describes what is available with BLISS-36 V3. Note that the BLISS tools (utility programs) and system interfaces described here are not supported products at the time of writing.

9.1 TRANSPORTABLE PROGRAMMING TOOLS (XPORT)

XPORT is a collection of transportable source-level programming tools for use with the BLISS language. XPORT tools may be commonly applied across all BLISS target systems to provide such things as: extensive input/output facilities; a uniform interface for obtaining operating system services (such as dynamic memory); and aids to data structuring and string handling.

The XPORT package consists of five components:

XPORT Data Structures
XPORT Input/Output Facilities
XPORT Dynamic Memory Management
XPORT Host System Services
XPORT String Handling Facilities

Each component provides tools which ease the task of interfacing a BLISS program with the operating system under which it will run. Therefore, the primary purpose of the XPORT package is to provide tools and interfaces which behave exactly alike in all system environments, and thus provide transportable operating system interfaces.

Programs written in Common BLISS (which use XPORT services in the manner prescribed) can be developed and debugged on one system and run on any other BLISS-supported system without change.

An additional benefit of using the XPORT package for BLISS programs, even if they do not require transportability, is the advantage in using the simplified XPORT interface as opposed to more powerful and complicated host system interfaces.

A description of each XPORT component follows; however, for a more detailed explanation of XPORT and its services refer to the XPORT Programmer's Guide.

9.1.1 XPORT Data Structures

The XPORT structure-definition facility is a collection of macros that allow a programmer to define efficient BLOCK structures in a manner that is both convenient and primarily system-independent. The facility primarily consists of a replacement for the standard BLISS FIELD declaration, using the keyword $FIELD.

The structure-definition facility allows a programmer to name the kind of field required for each block component, instead of specifying its position and size. A field-type (such as, SHORT_INTEGER, ADDRESS, BYTE) implies not only the size but also the alignment and required sign extension mode.

The XPORT data structure facility also provides the following support features:

FIELD_SET_SIZE	Calculates the size of a block defined by $FIELD.
ALIGN	Forces a specified mode of alignment for a subsequent field.
OVERLAY	Allows for overlayed field definitions.
CONTINUE	Terminates field overlaying.
LITERAL/DISTINCT	Creates a set of distinct integer literals.
SHOW	Controls the display of XPORT generated field definitions, values, and messages.
SUB_FIELD	Provides a means of referencing a field within a substructure of a block.

9.1.2 XPORT Input/Output

XPORT input/output is a general-purpose, system-independent service that supports sequential I/O operations in record, character stream, and binary mode, and provides basic file functions. This facility actually consists of several separate I/O packages, each being written for a specific operating system and file system. However, the program interface to each package is identical. Thus, a transportable I/O interface is provided for programs written in common BLISS or any other transportable language.

The XPORT I/O facility performs the following I/O and file manipulation functions:

OPEN	Prepares a file for reading (input) or writing (output). An output file may be optionally created.
CLOSE	Terminates the processing of an input or output file, including the flushing of any I/O buffers.
DELETE	Deletes an existing file.
RENAME	Changes the name of an existing file.

BACKUP	Provides a mechanism for preserving a copy of an input file when a program creates a new version of that file. This capability is typically used by editor-type applications.
PARSE	Parses a host system file specification into its component parts.
GET	Returns the length and address of the next sequential logical record read from an input file. Logical concatenation of several input files can be automatically performed when an intermediate end-of-file is reached.
PUT	Writes a single logical record into an opened output file.

9.1.3 XPORT Dynamic Memory Management

The XPORT dynamic memory management facility provides the following functions:

GET_MEM	Allocates a specified amount of dynamic memory.
FREE_MEM	Releases an allocated element of dynamic memory

9.1.4 XPORT Host System Services

The XPORT host system services are a set of routines that perform commonly needed host system functions in a transportable manner. The functions provided are as follows:

PUT_MSG	Routes a message sequence to the standard output and/or error devices, based on a message severity code.
TERMINATE	Terminates program execution after sending the user a termination message.

9.1.5 XPORT String Handling Facilities

The XPORT string handling facility provides a programmer with the ability to transportably manipulate character strings. Small control structures (modeled after the VAX/VMS descriptor convention) are used to facilitate the exchange of character data between procedures.

The following descriptor classes are provided:

FIXED	Describes a string with a fixed length and location.
BOUNDED	Describes a buffer that contains a variable length string.
DYNAMIC	Describes a moveable string, the length of which is subject to variance.
DYNAMIC_BOUNDED	Describes a moveable buffer that contains a variable length string.

The following functions are used to manipulate a descriptor and its associated string:

DESCRIPTOR	Creates and initializes a descriptor in OWN storage, or creates a descriptor in LOCAL storage.
DESC_INIT	Dynamically initializes a descriptor in OWN or LOCAL storage.
COPY	Copies a source string to a target string with appropriate truncation and padding.
APPEND	Appends a source string to a target string.
EQL,NEQ,LSS, LEQ,GEQ,GTR	Compares the values of two strings according to the ASCII collating sequence.
SCAN	Locates a specific sequence of characters (FIND mode), matches a stream of characters (SPAN mode), or searches for one character of a set (STOP mode).
CONCAT	Concatenates two or more strings as a single logical string.
FORMAT	Centers, left justifies, right justifies, or converts a string to upper case.
ASCII	Produces an ASCII string representation of a binary field value.
BINARY	Converts an ASCII string value to a binary value.

The XPORT string handling facility also extends the descriptor concept to include binary data; whereby, the four descriptor classes (FIXED, BOUNDED, DYNAMIC, and DYNAMIC_BOUNDED) are used to describe a data item instead of a string, while two of the descriptor manipulation functions (DESCRIPTOR, DESC_INIT) are used to create and intitialize the binary data descriptors.

9.2 BCREF - BLISS MASTER CROSS REFERENCE PROGRAM

The BCREF program is a global cross-reference utility that is used to merge cross-reference files (.CRF), for an entire system of modules, and create a master cross-reference listing file (.LIS).

9.2.1 Command-Line Format

BCREF command lines use the following syntax:

```
=>10

    .R BCREF
    *{output-spec}=input-spec,...{switch...}

=>20

    @BCREF
    BCREF>input-spec,...{/OUTPUT:output-spec}{switch...}
```

The default file extensions are:

input-spec:	.CRF
output-spec:	.LIS

Note that the input-list option is a comma list of cross-referenced input-file-specs (filename.CRF) to be merged.

Note that by default the output-file-name (filename.LIS) will be the same as that of the first input-file-name (filename.CRF) on the list.

9.2.2 Command Semantics

The master cross-reference output file (.LIS) provides a merge-module listing that is similar to the module-specific listing produced by the compiler (see Section 3.2.5). The differences are:

- The compiler listing header does not appear; however, the module-specific header does.
- The module name in which a symbol is declared appears after the symbol name.
- The size of the module name field reflects the longest possible module name in the merged system.

9.2.3 Building a Master Cross Reference

A master cross reference is an alphabetic listing of all user identifiers contained in a group of modules. For each identifier, the listing contains the names of modules and the line number within each module when that identifier occurs. You produce the listing by creating cross-reference files (.CRF) for each module and merging the files to produce a single output file (.LIS).

The following command sequences can be used to create master cross-reference file FILES.LIS.

```
=>10

     .R BCREF
     *FILES=FILEA,FILEB
     *^C

=>20

     @BCREF
     BCREF>FILEA,FILEB=FILES
```

9.2.4 Command Switches

The bcref-switches indicate what can be included or excluded from the master cross-reference listing (* = default).

/MULTIPLE /NOMULTIPLE*	Include all similar-type multiple references to symbols occurring on the same source line.
/HEADER* /NOHEADER	Include page header for the master cross-reference listing.

/LOG* /NOLOG	Provide informational messages relating to the BCREF data file merge process.
/OUTPUT:file-spec	Specifies output file for TOPS-20.

9.3 CVT10 - BLISS-10 TO BLISS-36 CONVERSION PROGRAM

CVT10 is a tool for converting BLISS-10 source text into BLISS-36. CVT10 is designed to do a large percentage of the syntax conversions and some smaller set of other conversions. Since CVT10 is not a compiler, it is limited in the completeness of translation as well as in the number of ways certain legal constructs can be written and still get transformed.

CVT10 assumes that the input compiles correctly with the BLISS-10 compiler. CVT10 is written in the SITBOL version of SNOBOL. This section describes the internal design of the CVT10 program to help fix a bug or add a transformation that will aid a conversion process.

9.3.1 CVT10 Command-Line Syntax

CVT10 prompts for both input files and outputs. CVT10 commences execution by typing:

```
INPUT FILE NAMES SEPARATED BY SPACES
```

At this point the user can enter input-file-specs. If no file-spec is specified, input is taken from the terminal. If more than one file-spec is specified, multiple translations occur.

CVT10 next prompts:

```
OUTPUT FILE NAMES
```

The user should now enter exactly as many output-file-specs as there were input-file-specs. The first input-file is translated and becomes the first output-file; the second input-file is translated and becomes the second output-file; and so on. If no file-spec is specified, the output goes to the terminal, in which case, input can come from either the terminal or a single input-file.

The file-specs have the following format:

```
{device:} filename.extension {[project,programmer-number]}
```

An example of a CVT10 command-line sequence follows.

```
RUN CVT10

INPUT FILE NAMES SEPARATED BY SPACES
MAIN.B10 SUBR.B10
OUTPUT FILE NAMES
MAIN.B36 SUBR.B36

INPUT FILE NAMES SEPARATED BY SPACES
^C
```

9.3.2 BLISS-10 Translations

CVT10 attempts to translate most constructs found in BLISS-10 programs. However, no conversion is attempted for some features of the BLISS-10 language. It is important to realize that CVT10 is sensitive to both the format and complexity of the source. Table 9-1 lists the language features of BLISS-10 in three categories, according to what degree CVT10 attempts conversion.

If CVT10 detects syntax that it cannot properly correct, it usually prints a warning to that effect. However, it is important to check over the translation for constructs that may cause problems.

Table 9-1: BLISS-10 Language Features

Conversion Attempted	No Conversion Needed	No Conversion Attempted
! and % comments	DO WHILE/UNTIL	LINKAGE
quoted string	INCR/DECR	SIGNAL/ENABLE
BIND	LABEL	PSECT/CSECT
EXTERNAL	LEAVE	NOVALUE
FORWARD		SCANx
GLOBAL		REPLACEx
GLOBAL BIND		COPY
GLOBAL ROUTINE		INCP
ROUTINE		FIRSTONE
LOCAL		OFFSET
MACRO		EXIT
MAP		GLOBALLY
MODULE		INDEXES
OWN		NAMES
REGISTER		
REQUIRE		
STACKLOCAL		
octal constants		
string type keywords		
[(]ASCII, ASCIZ, RADIX50,		)
SIXBIT[		]
quoted strings		
[(]'AB?M?J ->		)
%STRING[(]'AB',%CHAR(13),)		
%CHAR(10)[		][ ]
CASE		
SELECT		

When encountered, four special comments cause CVT10 to take special action:

!CVTCOM — Make all input lines that follow into comments. This is used to save the original BLISS-10 code as a comment when different code has been added.

!MOCTVC — Turn off the effect of !CVTCOM. Process source normally.

!CVTEXT Treat all input that follows as if it was BLISS-36 code that had been commented out. Remove the comment character (!) and output each line without further change.

!TXETVC Turn off the effect of !CVTEXT. Process source normally.

The following sections describe the restrictions associated with the various language feature conversions.

9.3.2.1 Normal Declarations - The following declarations are translated: EXTERNAL, FORWARD, GLOBAL, ROUTINE, GLOBAL ROUTINE, LOCAL, MAP, OWN, REGISTER, AND STACKLOCAL. Several identifiers may be declared in the same declaration. A declaration can span multiple lines. However, each identifier must have its whole declaration on a single line, or the translation will be done incorrectly.

A heuristic trick used when the end cannot be found is to check to see if there was an attempt to initialize the symbol. If so, 'INITIAL' is put out before the next line is processed.

Here is an example of the heuristic:

BLISS-10 Input

```
OWN X[5],Y,Z[3]= (2,
                  1,
                  6);
```

BLISS-36 Translation

```
OWN
   X: VECTOR[5],
   Y,
   Z: VECTOR[3] INITIAL (2,
                         1,
                         6);
```

9.3.2.2 REQUIRE Declarations - The file specifications must be on the same line as the 'REQUIRE' reserved word.

9.3.2.3 SWITCHES Declarations - The entire line on which any SWITCHES declaration occurs is removed from the program. Thus, to avoid error, the SWITCHES declaration must be completely contained on a single line. No other code can occur on the same line as a SWITCHES declaration.

9.3.2.4 BIND Declarations - BINDs to other than PLITS or UPLITS must be completely defined on a single line. Thus:

```
BIND DVSTRC Z = FRNAM;
```

translates properly, but the following does not:

```
BIND DVSTRC Z =
              FRNAM;
```

BINDs to PLITs and UPLITs can span many lines, but they must be the last definition in the BIND declaration. This means you can have only one PLIT binding in each BIND declaration.

9.3.2.5 ROUTINE Declarations - If a ROUTINE (or GLOBAL ROUTINE) declaration contains a linkage attribute, the following must be all on one line:

- ROUTINE (or GLOBAL ROUTINE) keyword(s)
- Linkage attribute
- Routine parameter list
- Equals sign

As an example, in BLISS-10

```
ROUTINE FORTRAN START (A,B,C)=
```

would be translated into BLISS-36 as:

```
ROUTINE START (A,B,C) : FORTRAN
```

but the following is invalid:

```
ROUTINE FORTRAN START
        (A,B,C) =
```

9.3.2.6 SELECT Expressions - CVT10 does not parse the SELECT expression. It simply makes note of what is going by on a line by line basis. After the NSET is encountered, each colon signals that the characters that precede it on that line constitute the SELECT labels. Nothing can precede a SELECT label on its line. The SELECT label must be on the same line as the colon that follows it.

Examples

BLISS-10

```
SELECT .X OF
    NSET
    3:    FOO(.K+1);
    #26:  .K;
    .V:   .Z*.K
    TESN;
```

BLISS-36

```
SELECT .X OF
    SET
    [3]:       FOO(.K+1);
    [%O'26']:  .K;
    [.V]:      .Z*.K
    TES;
```

9.3.2.7 CASE Expressions - The CASE expression should be formatted so that no more than one CASE label is required for each line.

9.3.2.8 MACROs - The closing $ in a macro is converted to %. An attempt is made to transform all occurrences of $ and % characters in a BLISS-10 macro. If a language construct is completely contained within a macro, it is normally transformed properly, as in:

```
MACRO A = OWN X[2] $;
```

However, macros that contain only portions of an expression, declaration, or statement are transformed incorrectly, as in:

```
MACRO B = OWN [ $;
```

9.4 TUTIO - TUTORIAL TERMINAL INPUT/OUTPUT PACKAGE

TUTIO.R36 is a BLISS REQUIRE file that contains some simple terminal I/O primitives. It is normally used in conjuction with the BLISS self paced study course as outlined in the BLISS Primer, but can be useful in writing quick and dirty programs, also. To gain access to the elements of this package, insert the following line in your BLISS program:

```
REQUIRE 'TUTIO';
```

A list of these primitives and their functions appears below. The following conventions are used in the descriptions:

```
char        - a character
len         - a length (in characters)
addr        - a memory address
value       - an integer
radix       - an 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.

9.5 SYSTEM INTERFACES

9.5.1 Precompiled Declaration Libraries

Precompiled BLISS declaration libraries are provided to facilitate the use of the TOPS-10 and TOPS-20 monitor calls in BLISS-36 programs. These libraries are derived from the MACRO-10 UNIVERSAL files:

- MACTEN.UNV
- MACSYM.UNV
- UUOSYM.UNV
- MONSYM.UNV

which are used in writing MACRO programs. The libraries which are provided are:

- TENDEF.L36 - Monitor independent symbols from MACTEN and MACSYM
- UUOSYM.L36 - TOPS-10 monitor calls from UUOSYM
- MONSYM.L36 - TOPS-20 monitor calls from MONSYM

Generally speaking, BLISS declaration libraries contain definitions of the same symbols which the MACRO universal files contain, and are used in much the same way as the universal files. More detail about how the translation occurs is found in later sections.

The symbol names are the same except that the period is translated to dollar sign, and the percent sign is translated to underline, e.g.,

```
PC%USR  becomes  PC_USR
.CHLFD  becomes  $CHLFD
```

This rule permits the programmer to use the published UUOSYM and MONSYM files as a guide to the use of the BLISS libraries, while the special character translation avoids the use of the %NAME lexical-function.

9.5.2 TENDEF.L36 Library

The TENDEF library contains a number of MACRO and LITERAL declarations which are not monitor dependent. These provide definitions of PDP-10 hardware data structures, and definitions needed to utilize the mask definitions contained in the UUOSYM and the MONSYM libraries.

Ordinarily both TENDEF and either UUOSYM or MONSYM will be needed:

```
LIBRARY 'TENDEF';
LIBRARY 'MONSYM';
```

9.5.2.1 POINTR Macro - Subfields of a machine word are defined in the libraries in terms of bit masks just as they are defined in the corresponding MACRO universal files. These masks are a contiguous string of one-bits arbitrarily located in a 36-bit word. The POINTR macro is used to construct a field-reference from these masks.

Call:

```
POINTR ( address , mask )
```

Expansion:

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

Example:

```
POINTR(X, %O'60') expands to X<4,2>
```

9.5.2.2 FLD Macro - The FLD macro is used to position a value in a subfield specified by a bit mask.

Call:

```
FLD ( value , mask )
```

Expansion:

```
( ( value ) Position )
```

Example:

```
FLD(2, %O'60') expands to ((2) 4)
```

9.5.2.3 MONWORD and MONBLOCK Structures - Since subfields of a machine word are defined in MONSYM and UUOSYM as bit masks, the MONWORD and MONBLOCK structures are defined to accept a mask value to determine position and size.

For example:

```
OWN
    X:MONWORD;
  .
  .
  .
X[PC_OVF]= 0;  !PC_OVF is defined as 1^35
```

is equivalent to:

```
X<35,1>= 0;
```

The MONBLOCK structure defines a BLOCK of words, each of which is accessed as a MONWORD:

```
LOCAL
    V:MONBLOCK[8];
  .
  .
  .
INCR I FROM 0 TO %ALLOCATION(V)-1 DO
     V[.I,%O'777777']= 0;
```

The latter would deposit zeros into:

```
(V+.I)<0,18>= 0
```

9.5.2.4 Other Symbols - The TENDEF library contains several other groups of symbols extracted from the MACSYM file:

```
Program counter bits  PC_xxx
Version number subfields  VI_xxx
Control character definitions  $CHxxx
```

9.5.3 UUOSYM.L36 Library

The following specifies the translation of each type of symbol defined by the UUOSYM file into BLISS:

- Symbols defined by equates, which include offsets, field values, and subfield bit masks, are converted to LITERAL declarations with the same value.

- Symbols defined by OPDEF, which include the UUO instructions, are converted to MACRO declarations. One of four styles of macro is chosen depending upon the form of the OPDEF. In each case, the macro ultimately expands after parameter substitution to a list of three values separated by commas, which represent the operation code, the accumulator field, and the effective address of the instruction. This macro would ordinarily be called to supply the parameters of a MACHOP or MACHSKIP function.

  ```
  MACRO name= %O'xxx', %O'xx', %O'xxxxxx' %;
  ```

  This form is used when the accumulator field and the effective address of the OPDEF are both nonzero.

  ```
  MACRO name(a)= %O'xxx', a, %O'xxxxxx' %;
  ```

  This form is used when the accumulator field of the OPDEF is zero and the effective address is nonzero, or when the opcode is CALLI.

  ```
  MACRO name(e)= %O'xxx', %O'xx', e %;
  ```

  This form is used when the accumulator field of the OPDEF is nonzero and the effective address is zero, or when the opcode is TTCALL or MTAPE.

  ```
  MACRO name(a,e)= %O'xxx', a, e %;
  ```

  This form is used when the accumulator field and the effective address of the OPDEF are both zero (and the opcode is not one of the special cases).

9.5.4 MONSYM.L36 Library

The following specifies the translation of each type of symbol defined by the MONSYM file into BLISS:

- Symbols defined by equates, which include offsets, field values, JSYS error codes, and subfield bit masks, are converted to LITERAL declarations with the same value.

- Symbols defined by OPDEF, which include the JSYS instructions, are converted to LITERAL declarations with the JSYS number as value. The symbol "JSYS" itself is omitted.

9.5.5 Generation Procedure

The following is the procedure for library generation. The inputs to the generation are:

```
UUOSYM.MAC
MONSYM.MAC
TENDEF.R36    Monitor independent symbols
FLDBB.R36     FLDDB$ macro

.R MACRO           ! MACRO 52 or later
*,UUOSYM=UUOSYM      ! Make UUOSYM.LST
*,MONSYM=MONSYM      ! Make MONSYM.LST
.RUN MONINT      ! Convert .LST to .R36
*UUOSYM            ! Make UUOSYM.R36
*MONSYM            ! Make MONSYM.R36
.R BLISS           ! Create libraries
*TENDEF=TENDEF/LIBRARY     ! Make TENDEF.L36 from TENDEF.R36
*UUOSYM=UUOSYM/LIBRARY     ! Make UUOSYM.L36 from UUOSYM.R36
*MONSYM=MONSYM,FLDDB/LIBRARY  ! Make MONSYM.L36 from MONSYM.R36
                    ! and FLDDB.R36
.DELETE UUOSYM.LST,MONSYM.LST,UUOSYM.UNV,MONSYM.UNV
```

The following files should now be installed in a public directory:

```
TENDEF.R36  TENDEF.L36
UUOSYM.R36  UUOSYM.L36
MONSYM.R36  MONSYM.L36
FLDDB.R36
```

9.5.6 TOPS-10 System Interface Example

```
MODULE tug10( %TITLE'Bliss-36 TOPS-10 Interface Example'VERSION='1(1)',
                MAIN=show_time, ENVIRONMENT(TOPS10))=
BEGIN

!++
!
! FUNCTION
!
!       Simple example showing interfacing techniques for Bliss-36
!       and TOPS-10 V7.01.
!
!--

FORWARD ROUTINE
    show_time    : NOVALUE,
    gettime      : NOVALUE,      ! Loads buffer with ASCIZ time string.
    outdec       : NOVALUE;      ! Decimal conversion routine

LIBRARY 'BLI:TENDEF';

LIBRARY 'BLI:UUOSYM';

LITERAL
    LHALF       = -1^18,                ! Masks suitable for use with
    RHALF       = %O'777777';           !  MONWORD structure.

MACRO
    GETLCH_UUO(arglist) =       UUO(0, GETLCH(arglist)) %,
    GETTAB_UUO(argadr)  =       UUO(1, GETTAB(argadr)) %,
    OUTSTR_UUO(stradr)  =       UUO(0, OUTSTR(stradr)) %;
```

```
BUILTIN
    UUO;

MACRO
    MSGL[] = MSG( %REMAINING, %CHAR($CHCRT, $CHLFD) ) %,
    MSG[] = UPLIT( %ASCIZ %STRING( %REMAINING ) ) %;

ROUTINE show_time : NOVALUE=
!+
! FUNCTION
!       Simple program which gets current date-time info (via GETTAB)
!       and displays a text string on the user's TTY:
!-

    BEGIN
    LOCAL
        timebuffer      : VECTOR[CH$ALLOCATION(80)],

        line_status     : MONWORD INITIAL(-1);  ! Initial value for job's
                                                !  controlling terminal

! Find out what kind of terminal the controlling job has
! and print some BELs unless it's a PTY:

GETLCH_UUO( line_status );

IF .line_status[LHALF] EQL 0
THEN
   RETURN OUTSTR_UUO( MSGL('GETLCH failed') );

IF NOT .line_status[GL$ITY]                 ! PTYs don't get "BELLS"
THEN
   OUTSTR_UUO( MSG( %CHAR($CHBEL, $CHBEL, $CHBEL) ) );

gettime( CH$PTR(timebuffer) );

OUTSTR_UUO( MSG('%SHOW_TIME:: '));
OUTSTR_UUO( CH$PTR(timebuffer) );
OUTSTR_UUO( MSG(%CHAR($CHCRT, $CHLFD)) )
END;
ROUTINE gettime( bufptr ) :NOVALUE =
!+
! FUNCTION
!       Get the current date-time from the monitor, convert to ASCIZ
!       of the form
!
!               dd-mmm-yyyy  hh:mm:ss
!
! INPUT
!       bufptr  - CH$PTR to output buffer
!
! OUTPUTS
!       None
!-
    BEGIN
    REGISTER
        argval  : MONWORD;      ! Needed for doing GETTABs

    BIND
        tabvals = PLIT( _CNDAY, _CNMON, _CNYER, _CNHOR, _CNMIN, _CNSEC )
                        : VECTOR,

        monstr = UPLIT( '-Jan-', '-Feb-', '-Mar-', '-Apr-',
                        '-May-', '-Jun-', '-Jul-', '-Aug-',
                        '-Sep-', '-Oct-', '-Nov-', '-Dec-' ): VECTOR;
```

```
    LOCAL
        timvals : VECTOR[6]; ! Holds results of GETTABs

    INCR i FROM 0 TO .tabvals[-1] - 1 DO
        BEGIN
        argval[LHALF] = .tabvals[.i];    ! Item from System Config Table
        argval[RHALF] = $GTCNF;

        IF NOT GETTAB_UUO( argval )
        THEN                             ! The UUO failed!
         RETURN OUTSTR_UUO( MSGL('?GETTAB failed') );

        timvals[.i] = .argval            ! Copy into safe place
        END;

    ! Now write time values as needed.
    !

    OUTDEC( .timvals[0], bufptr );

    bufptr = CH$MOVE( 5, CH$PTR(monstr[.timvals[1]]), .bufptr );

    OUTDEC( .timvals[2], bufptr );

    CH$WCHAR_A( %C' ', bufptr );

    OUTDEC( .timvals[3], bufptr );       ! HH:
    CH$WCHAR_A( %C':', bufptr);

    OUTDEC( .timvals[4], bufptr );       ! MM:
    CH$WCHAR_A( %C':', bufptr );

    OUTDEC(.timvals[5], bufptr );        ! SS
    CH$WCHAR_A( 0, bufptr );             ! The trailing NUL for ASCIZ strings
    END;
ROUTINE OUTDEC( value, bufaddr_adr ): NOVALUE=
!+
! FUNCTION
!       Convert a number from binary to ascii decimal representation
!
! INPUTS
!       value - numeric value to convert
!       bufaddr_adr - address of CH$PTR to use. (Incremented as we go)
!
! OUTPUTS
!       none
!
! NOTES
!       Only positive numbers are handled.
!-
    BEGIN
    IF .value LEQ 9
    THEN
        CH$WCHAR_A( .value+%C'0', .bufaddr_adr )
    ELSE
        BEGIN
        OUTDEC( .value/10, .bufaddr_adr );
        OUTDEC( .value MOD 10, bufaddr_adr )
        END
    END;

END ELUDOM
```

9.5.7 TOPS-20 System Interface Example

```
MODULE tug (MAIN=showtime, version='1(1)' %TITLE'BLISS-36 User''s Guide')=
BEGIN

!++
!       Simple program to demonstrate using JSYS in Bliss-36
!
!--

FORWARD ROUTINE
    readaline,                  ! Read one line from the terminal
    show_time   : NOVALUE,      ! Print full date and time
    cmdmsg      : NOVALUE;      ! Print a message on the terminal

LIBRARY 'SYS:TENDEF';

UNDECLARE
        $CHCRT,         ! These are declared in both libraries...
        $CHLFD;         !               "

LIBRARY 'SYS:MONSYM';

MACRO
    ! Given JSYS numbers from MONSYM, create linkages and BIND ROUTINE
    ! declarations to define a more esthetic interface to TOPS-20 JSYS.
    !

    MJSYS(name,skipcnt,inreg,outreg)=
        %ASSIGN(jsysno,name)
        UNDECLARE name;

        LINKAGE %NAME('1_',name) = JSYS (
                %IF NOT %NULL(inreg) %THEN RPLIST( %REMOVE(inreg) ) %FI
                %IF NOT %NULL(outreg) %THEN ; RPLIST( %REMOVE(outreg) ) %FI)

                : SKIP(skipcnt);

        BIND ROUTINE name = jsysno : %NAME('1_',name); %,

     RPLIST(a)[] = REGISTER=a%IF %LENGTH GTR 1
                            %THEN , RPLIST(%REMAINING) %FI %;

     COMPILETIME
         jsysno = 0;

%(              Name  Skip  In-Regs     Out-Regs )%
%(              ----  ----  -------     -------- )%

     MJSYS(     DVCHR, 0,   (1),        (1,2,3)  );
     MJSYS(     PSOUT, 0,   (1)                  );
     MJSYS(     ODTIM, 0,   (1,2,3)              );
     MJSYS(     RDTTY, 1,   (1,2,3),    (2)      );

BIND
    CRLF = CH$PTR( UPLIT( %ASCIZ %CHAR($CHCRT, $CHLFD) ) );

GLOBAL ROUTINE cmdmsg( amsg )           : NOVALUE=

!++
! FUNCTION
!       Print out a string on the primary output device.
!
```

```
! INPUTS
!       amsg - CH$PTR to an ASCIZ string
!
! OUTPUTS
!       None
!--

    BEGIN

    LOCAL
        characteristics : MONWORD;

    ! Get device characteristics to see if we should give a CRLF
    !

    DVCHR( $PRIOU ; , characteristics );

    IF .characteristics[DV_TYP] EQL $DVTTY OR    ! Real TTY or
       .characteristics[DV_TYP] EQL $DVPTY       !  PTY: get a CRLF
    THEN
        PSOUT( CRLF );

    PSOUT( .amsg );             ! Write the message text
    PSOUT( CRLF )               !  and a CRLF afterwards
    END;

ROUTINE show_time : NOVALUE=
!++
! FUNCTION
!       Print the current time-of-day in the format
!
!               hh::mm::ssAM-EDT
!
! INPUTS
!      None
!
! OUTPUTS
!      None
!--
    BEGIN
    LITERAL
        buflen = 133;
    LOCAL
        buffer : VECTOR[ CH$ALLOCATION(buflen) ];

    ODTIM( CH$PTR(buffer), -1, OT_NDA+OT_12H+OT_SCL );

    cmdmsg( CH$PTR(buffer) );

    IF readaline( CH$PTR(buffer), buflen, CH$PTR(UPLIT(%ASCIZ'GOOD?') ) )
    THEN
        ! Successful
        !
        cmdmsg( CH$PTR( UPLIT( %ASCIZ'BYE' ) ) )
    ELSE
        cmdmsg( CH$PTR( UPLIT( %ASCIZ'FAILED' ) ) )

     END;               ! show_time

ROUTINE readaline ( bufadr, len, prompt ) =
!+
! FUNCTION
!       Accept a line from the terminal
```

```
! INPUTS
!       bufadr  - CH$PTR to buffer where terminal response should be placed
!       len     - length of buffer
!       prompt  - CH$PTR to ASCIZ string to prompt the user with.
! OUTPUTS
!       1       We got a line of input
!       0       It flopped.
!-

    BEGIN

    PSOUT( .prompt );

    ! RDTTY skip-return is converted to TRUE or FALSE
    !

    RDTTY( .bufadr, RD_TOP+.len, .prompt )
    END;

END ELUDOM
```

CHAPTER 10

BLISS-36 CODING EXAMPLES

This chapter provides useful examples of BLISS-36 coding techniques. The coding used to devise these examples demonstrates the use of many BLISS language features.

10.1 EXAMPLE 1: THE PSINT PROGRAM

The coding of the PSINT program primarily demonstrates the use of the PSI_INTERRUPT linkage. The program allows you to display a specified file in 132 columns on the terminal. A Control-C is used to both exit the program and return the terminal width to its original setting.

The PSINT program requires the following system interface files:

- MONSYM - TOPS-20 monitor interface symbols
- TENDEF - Monitor independent MACRO and LITERAL declarations
- MJSYS - Interface to TOPS-20 JSYS calls

The following steps are used to run the PSINT program:

1. Compile the PSINT program:

```
@BLISS
BLISS>PSINT
; File:           PS:<JONES>PSINT.B36.1
; Size:           328  code + 2123 data words
; Run Time:          00:05.2
; Elapsed Time:      00:16.7
; Lines/CPU Min:      4450
; Lexemes/CPU-Min:   74859
; Memory Used:  39 pages
; Compilation Complete
BLISS>^C
@
```

2. Link the PSINT program and use /GO to terminate:

```
@LINK
*PSINT
*/GO

EXIT
```

3. Save the executable image:

```
@SAVE PSINT
PSINT.EXE.1 Saved
```

4. Run the PSINT program:

```
@RUN PSINT
(terminal is set to 132 columns)
FILE:ALPHA.TXT
(ALPHA.TXT is displayed in 132 columns)
FILE:^C
@(terminal is reset to original width)
```

The following sections contain the annotated coding of the PSINT module:

10.1.1 Module PSINT

```
MODULE PSINT( %TITLE'Bliss-36 PS_INTERRUPT Example'
VERSION='1(1)',
              MAIN=REPLAY )=
BEGIN

!++
!
! FUNCTION
!
!     Example showing the use of the PS_INTERRUPT linkage
!     declaration in Bliss-36.
!
!     This example types a specified file to the TTY
!     in 132 columns. Control-C must be typed in order
!     to exit, causing the terminal width to be reset
!     to its original setting. If a Control-Y is typed
!     the program will prompt for another file.
!
!--

LINKAGE PSI = PS_INTERRUPT;

FORWARD ROUTINE

    REPLAY  :  NOVALUE,
    DISPLAY :  NOVALUE,
    ENAPSI  :  NOVALUE,      ! Sets up and enables the PSI
    CTRCLC  :  PSI NOVALUE,  ! The Control-C software interrupt
                             ! handler
    CTRLY   :  PSI NOVALUE,  ! The Control-Y software interrupt
                             ! handler
    TTYSET  :  NOVALUE,      ! Sets up terminal characteristics
    FILIO   :  NOVALUE,      ! Performs file I/O
    TTYRES  :  NOVALUE,      ! Resets terminal characteristics
    DISPSI  :  NOVALUE,      ! Disables the PSI
    DSPHDL,                  ! Condition Handler for the DISPLAY
                             ! routine
    PSIHDL;                  ! Condition Handler for interrupt
                             ! routine

LIBRARY 'BLI:TENDEF';

UNDECLARE

    $CHCRT,                  ! Declared in both libraries
    $CHLFD;

LIBRARY 'BLI:MONSYM';

REQUIRE 'MJSYS';             ! Define needed JSYS calls
```

```
LITERAL

    true = 1,
    false = 0,
    ss$cntrl_y = 1,          ! Control-Y flag
    intchl = 1,              ! Interrupt channel 1
    intch2 = 2,              ! Interrupt channel 2
    bufsiz = 132,            ! Size of the file I/O buffer
    typwid = 132;            ! Terminal output width

MACRO

    ERROR = ( ERSTR($PRIOU, ($FHSLF^18) OR %O'777777', 0);  (1)
                    HALTF(); )%,


    XPDP10_BITS[BITS] = 1^(35-BITS)%,

    PDP10_BITS[] = (0 OR XPDP10_BITS(%REMAINING))%;          (2)

OWN
    pclev1,                                  (3)
    levtab :  VECTOR[3]                      (4)
      INITIAL(pclev1, REP 2 OF (0)),

    chntab :  VECTOR[36]                     (5)
     INITIAL(0,(1^18 OR CTRLC),
                (1^18 OR CTRLY),
                 REP 33 OF (0)),
    filbuf :                                 (6)
        VECTOR[CH$ALLOCATION(bufsiz)],
    filjfn :  VOLATILE,                      (7)
    ttyjfn :                                 (8)
        VOLATILE INITIAL(-1),
    width  :  VOLATILE;                      (9)
```

1. The ERROR macro outputs the most recent error message to the primary output device and halts the process.

2. The PDP10_BITS macro returns a literal with the specified PDP-10 bits set.

3. PCLEV1 is the address used to save the PC during interrupts.

4. LEVTAB is the priority level table used by PSI.

5. CHNTAB is the channel table.

   Note that Channel 1 (priority level 1) serves the CTRLC routine, while Channel 2 (priority level 1) serves the CTRLY routine; no other channels are assigned.

6. FILBUF is the input file buffer.

7. FILJFN is the JFN for the file.

8. TTYJFN is the JFN for the terminal.

9. WIDTH is the original setting of the terminal.

10.1.2 Routine REPLAY

```
ROUTINE REPLAY :  NOVALUE =
!+
! FUNCTION
!
!       This is the main routine which causes the program
!       to continually loop until a Control-C is typed or
!       an error occurs.
!-

    BEGIN

    WHILE true DO DISPLAY();

    END;
```

10.1.3 Routine DISPLAY

```
ROUTINE DISPLAY :  NOVALUE =
!+
! FUNCTION
!
!       This routine enables a condition handler
!       and calls all the routines which perform
!       the file and terminal I/O.
!-

    BEGIN

    ENABLE DSPHDL;

    ENAPSI();       (1)

    TTYSET();       (2)

    FILIO();        (3)

    TTYRES();       (4)

    DISPSI();       (5)

    END;
```

1. Enable the PSI to trap on Control-C and Control-Y.
2. Set the terminal width to 132.
3. Get the file and display it on the terminal.
4. Set the terminal width back to the original setting.
5. Disable the PSI.

10.1.4 Routine ENAPSI

```
ROUTINE ENAPSI :  NOVALUE =
!+
! FUNCTION
```

```
!
!       Enables the software interrupt system and sets up
!       the program to trap on Control-C and Control-Y.
!-

    BEGIN

    SIR($FHSLF, ((levtab^18) OR chntab));      (1)
    EIR($FHSLF);                               (2)
    AIC($FHSLF, PDP10_BITS(intch1,intch2));    (3)
    ATI($TICCC^18 OR intch1);                  (4)
    ATI($TICCY^18 OR intch2);                  (5)
    END;
```

1. Specify the PSI table.

2. Enable the PSI.

3. Activate the interrupt channel.

4. Trap on Control-C.

5. Trap on Control-Y.

10.1.5 Routine TTYSET

```
ROUTINE TTYSET : NOVALUE =

!+
! FUNCTION
!
!       Opens the terminal in Image mode and sets
!       the width to 132, saving the original
!       setting.
!-

    BEGIN

    IF NOT GTJFN(GJ_SHT, CH$PTR(UPLIT(%ASCIZ'TTY:'));
                 ttyjfn)                                    (1)
    THEN
        ERROR;

    IF NOT OPENF(.ttyjfn, ( FLD(7,OF_BSZ) OR FLD(%O'10',
                 OF_MOD) OR OF_WR ))                        (2)
    THEN
        ERROR;

    MTOPR(.ttyjfn, $MORLW;,,width);       (3)

    IF .width NEQ typwid
    THEN
        BEGIN

        MTOPR(.ttyjfn, $MOSLW, typwid);  (4)

        SOUT(.ttyjfn, CH$PTR(UPLIT        (5)
            (%ASCII %STRING(%CHAR
            (%CHESC),'[?31'))), 5, 0);
        END
    END;
```

1. Associate a Job File Number with the terminal.

2. Open the terminal in image mode with a byte size of seven.

3. Get the terminal page width.

4. Set the software page width to 132.

5. The <ESC>[?3h' string is a special escape sequence that tells the VT100 to set the page width to 132.

10.1.6 Routine FILIO

```
ROUTINE FILIO :  NOVALUE =
!+
! FUNCTION
!
!       This routine prompts the user for a file name, opens
!       the specified file, and outputs it to the terminal.
!-

    BEGIN

    LITERAL
        max_spec = 219;
    LOCAL
        count,
        cond :  MONWORD,
        ttybuf :  VECTOR[ CH$ALLOCATION(max_spec) ];

    BEGIN

    PSOUT( CH$PTR(UPLIT(%ASCIZ'FILE:')) );

    IF NOT RDTTY(CH$PTR(ttybuf), max_spec, 0)       (1)
    THEN
        ERROR;

    IF NOT GTJFN((GJ_OLD OR GJ_SHT),CH$PTR(ttybuf);
                 filjfn)                            (2)
    THEN
        ERROR;

    IF NOT OPENF(.filjfn, (FLD(7,OF_BSZ) OR OF_RD)) (3)
    THEN
        ERROR;
    END;

    WHILE true DO
        BEGIN

        IF NOT SIN(.filjfn,                         (4)
         CH$PTR(filbuf),-bufsiz,0;,,count)
        THEN
            BEGIN
            GETER($FHSLF;cond);                     (5)
            IF .cond[IOX4]                          (6)
            THEN
                BEGIN
                IF .COUNT NEQ 0                     (7)
                THEN
                  SOUT(.ttyjfn,
                    CH$PTR(filbuf),(bufsiz-.count),0);
                EXITLOOP;
```

```
            END;
        ERROR                                          (8)
        END;
    SOUT(.ttyjfn,                                      (9)
     CH$PTR(filbuf),(bufsiz-.count),0);
    END;
END;
```

1. Read the file name from the terminal.
2. Associate a Job File Number with the file.
3. Open the file in 7-bit mode for reading.
4. Read from the input file.
5. Determine the cause of the error.
6. Stop reading when end-of-file is reached.
7. Clear the buffer.
8. Report any unexpected error condition.
9. Output the string to the terminal.

10.1.7 Routine TTYRES

```
ROUTINE TTYRES :  NOVALUE =
!+
! FUNCTION
!
!       Resets the terminal width back to the original
!       setting and closes and releases the JFN.
!-

    BEGIN

    IF (.width NEQ -1) AND                  (1)
       (.width NEQ typwid)
    THEN
        BEGIN

        MTOPR(.ttyjfn,                     (2)
         $MOSLW,.width);

        SOUT(.ttyjfn, CH$PTR(UPLIT         (3)
            (%ASCII %STRING(%CHAR
            (%CHESC),'[?31'))), 5, 0);
        width = 0;
        END;

    IF .ttyjfn GEQ 0
    THEN
        BEGIN

        CLOSF(.ttyjfn);
        ttyjfn = 0;
        END;

    END;
```

1. Check for altered terminal width.

2. Reset the terminal page width to its original setting.

3. Set the VT100 to the same width.

10.1.8 Routine DISPSI

```
ROUTINE DISPSI :  NOVALUE =
!+
! FUNCTION
!
!       Disables the software interrupt system.
!-

    BEGIN

    DTI($TICCY);                                    (1)
    DTI($TICCC);                                    (2)
    DIC($FHSLF, PDP10_BITS(intch1,intch2));         (3)
    DIR($FHSLF);                                    (4)
    END;
```

1. Disable trapping on Control-Y.

2. Disable trapping on Control-C.

3. Deactivate the interrupt channels.

4. Disable the PSI.

10.1.9 Routine CTRLC

```
ROUTINE CTRLC :  PSI NOVALUE =
!+
! FUNCTION
!
!       This is the Control-C interrupt handler. The handler
!       calls the routine to reset the width of the terminal
!       and then exits.
!
!       If CONTINUE is typed by the user, the terminal is
!       reset to 132 columns and the program continues from
!       the point where the trap occurred.
!-

    BEGIN

    TTYRES();

    HALTF();

    TTYSET();

    END;
```

10.1.10 Routine CTRLY

```
ROUTINE CTRLY :  PSI NOVALUE =
!+
! FUNCTION
!
!       This is the Control-Y interrupt handler. This
!       routine signals the PSI condition handler.
!-

    BEGIN

    ENABLE PSIHDL;       (1)

    SIGNAL(ss$cntrl_y);  (2)

    END;
```

1. Enable the PSI-handler.

2. Generate the Control-Y signal.

   Note that for a PSI-handler in which an UNWIND can occur, a condition handler must be established.

10.1.11 Routine DSPHDL

```
ROUTINE DSPHDL(sig: REF VECTOR, mech : REF VECTOR, enab : REF
VECTOR) =
!+
! FUNCTION
!
!       Condition handler for the routine Display. This
!       routine terminates all I/O and does an Unwind for
!       the Control-Y signal.
!-

    BEGIN

    IF .sig[1] NEQ ss$cntrl_y
    THEN
        RETURN false;         ! Resignal for conditions
                              ! other than Control-Y.
    TTYRES();

    CLOSF(-1);

    DISPSI();

    SETUNWIND();
    true

    END;
```

10.1.12 Routine PSIHDL

```
ROUTINE PSIHDL(sig: REF VECTOR, mech : REF VECTOR, enab : REF
VECTOR) =
```

```
!+
! FUNCTION
!
!       Condition handler for Control-Y. This routine
!       simply resignals.
!-

    BEGIN

    RETURN false

    END;


END
ELUDOM
```

10.2 EXAMPLE 2: THE TRANS PROGRAM

The coding of the TRANS program primarily demonstrates the use of the COMND JSYS function; however, the program also demonstrates BLISS-36 character-handling techniques. The program allows you to substitute or delete selected characters in an input file and produce an output file containing the changes.

The command line syntax is as follows:

```
in-spec {/OUTPUT:out-spec} (FROM) "characters" (TO) "characters"
```

The TRANS program requires the following system interface files:

- MONSYM - TOPS-20 monitor interface symbols
- TENDEF - Monitor independent MACRO and LITERAL declarations
- MJSYS - Interface to TOPS-20 JSYS calls

The following steps are used to run the TRANS program:

1. Compile the TRANS program:

```
@BLISS
BLISS>TRANS
; File:             PS:<JONES>TRANS.B36.1
; Size:             415  code + 2444 data words
; Run Time:            00:11.8
; Elapsed Time:        00:28.9
; Lines/CPU Min:        4771
; Lexemes/CPU-Min:    103118
; Memory Used:  54 pages
; Compilation Complete
BLISS>^C
@
```

2. Link the TRANS program and use /GO to terminate:

```
@LINK
*TRANS
*/GO
EXIT
```

3. Save the executable image:

```
@SAVE TRANS
TRANS.EXE.1 Saved
```

4. Run the TRANS program:

```
@RUN TRANS
TR>TRIN.TXT/OUTPUT:TOUT.TXT.1<ESC> !New file! (FROM) "CD"<ESC>(TO) "34"
@
```

Using the /OUTPUT: switch option, the program reads input file TRIN.TXT and creates output file TOUT.TXT in which all uppercase Cs and Ds are respectively changed to 3s and 4s. Note that if the /OUTPUT: switch is not used, an output file is created that is the next generation of the input file spec.

The following sections contain the annotated coding of the TRANS module.

10.2.1 Module TRANS

```
MODULE TRANS (MAIN = transmain
                  ) =
BEGIN

!++
! FUNCTION
!
!     This program copies an input file to an output file
!     with substitution or deletion of selected characters.
!
!     Each character in the argument (FROM) is translated to
!     the corresponding character in the argument (TO); all
!     other characters are copied as is.
!
!     Both (FROM) and (TO) may contain substrings of the form
!     a1-a2, as shorthand for all the characters in the range
!     a1..a2. If the (TO) argument contains no characters,
!     then all characters in the (FROM) argument are deleted.
!     If (FROM) is shorter than (TO), all characters in (FROM)
!     that would map to or beyond the last character in (TO)
!     are mapped to the last character in (TO).
!
!--

LIBRARY 'BLI:TENDEF';

UNDECLARE

    $chcrt,             ! These are declared in both libraries.
    $chlfd;

LIBRARY 'BLI:MONSYM';

REQUIRE 'MJSYS';        ! Define needed JSYS calls

MACRO                                                   (1)
    jsys_detected_error =
        ( erstr($priou, ($fhslf^18) OR %O'777777', 0);
                    haltf(); )%,
```

```
    quote_error =                                          (2)
        ( psout( CH$PTR(UPLIT(%ASCIZ'Invalid quoted string')));
                 haltf(); )%;

BIND
    switch = UPLIT (%ASCIZ'OUTPUT:');

LITERAL
    true = 1,
    false = 0,
    alpha = 1,
    alpha_lower_case = 2,
    numeric = 3,
    not_alpha_numeric = 4,
    buffer_length = 132;

OWN
    injfn,                          ! Input JFN
    outjfn,                         ! Output JFN
    state,                          ! Command state indicator
    reparse,                        ! Flag indicating a reparse
    eoc_seen,                       ! End of command flag
    cmdgjb : VECTOR [%O'16'],                              (3)
    swblk : VECTOR [2]                                     (4)
      INITIAL (1^18 OR 1, switch^18),
    command_buffer :                                       (5)
      VECTOR [CH$ALLOCATION (buffer_length)],
    atom_buffer :                                          (6)
      VECTOR [CH$ALLOCATION (buffer_length)],
    file_spec :                                            (7)
      VECTOR [CH$ALLOCATION (buffer_length)],
    from_count,                                            (8)
    from_buffer :                                          (9)
      VECTOR [CH$ALLOCATION (buffer_length)],
    to_count,                                              (10)
    to_buffer :                                            (11)
      VECTOR [CH$ALLOCATION (buffer_length)];
```

1. The JSYS_DETECTED_ERROR outputs the most recent error message to the primary output device and halts the process.

2. The QUOTE_ERROR outputs the "Invalid quoted string" message to the primary output device and halts the process.

3. CMDGJB is the argument block used by the COMND JSYS.

4. SWBLK is used by the COMND JSYS for parsing the /OUTPUT: switch

5. COMMAND_BUFFER is the text buffer used by the COMND JSYS.

6. ATOM_BUFFER is the atom buffer used by COMND JSYS.

7. FILE_SPEC is the buffer for the default output file.

8. FROM_COUNT indicates the number of characters in the FROM_BUFFER.

9. FROM_BUFFER is the character buffer for the (FROM) request.

10. TO_COUNT indicates the number of characters in the TO_BUFFER.

11. TO_BUFFER is the character buffer for the (TO) request.

The following chart depicts the command state table used by this example. Note that you initialize the BLOCKVECTOR state_table to reflect the entries to the table. The NEXT field contains the next state number, while the ACT field is used in a CASE statement to indicate what action is performed.

The following abbreviations define the actions performed:

```
s_i_i = save_input_info
s_f_p = save_from_part
s_o_j = save_out_jfn
s_i_j = save_in_jfn
s_t_p = save_to_part
```

Table 10-1: Depiction of the Command State Table

STATE	.CMINI (INIT)	.CMSWI (/OUTP)	.CMIFI (INPUT)	.CMOFI (OUTPUT)	.CMQST (QUOTES)	.CMCFM (EOC)	.CMNOI (FROM)	.CMNOI (TO)
0	Next=1	x	x	x	x	x	x	x
0	Act=null	x	x	x	x	x	x	x
1	x	Next=3	Next=2	x	x	x	x	x
1	x	Act=null	Act=s_f_p	x	x	x	x	x
2	x	Next=4	x	x	Next=8	x	x	x
2	x	Act=null	x	x	Act=s_f_p	x	x	x
3	x	x	x	Next=5	x	x	x	x
3	x	x	x	Act=s_o_j	x	x	x	x
4	x	x	x	Next=6	x	x	x	x
4	x	x	x	Act=s_o_j	x	x	x	x
5	x	x	Next=6	x	x	x	x	x
5	x	x	Act=s_i_j	x	x	x	x	x
6	x	x	x	x	x	x	Next=7	x
6	x	x	x	x	x	x	Act=null	x
7	x	x	x	x	Next=8	x	x	x
7	x	x	x	x	Act=s_f_p	x	x	x
8	x	x	x	x	x	x	x	Next=9
8	x	x	x	x	x	x	x	Act=null
9	x	x	x	x	Next=10	x	x	x
9	x	x	x	x	Act=s_t_p	x	x	x
10	x	x	x	x	x	Next=0	x	x
10	x	x	x	x	x	Act=eoc	x	x

```
LITERAL

    s0 = 0,                                         (1)
    s1 = 1,
    s2 = 2,
    s3 = 3,
    s4 = 4,
    s5 = 5,
    s6 = 6,
    s7 = 7,
    s8 = 8,
    s9 = 9,
    s10 = 10,

    cmini = 0,              ! Init                  (2)
    cmswi = 1,              ! Switch
    cmifi = 2,              ! Input file-spec
    cmofi = 3,              ! Output file-spec
    cmqst = 4,              ! Quoted string
    cmcfm = 5,              ! Confirm (crlf)
    cmnoi_from = 6,         ! Guide word (FROM)
    cmnoi_to = 7,           ! Guide word (TO)

    null = 0,                                       (3)
    save_input_info = 1,
    save_in_jfn = 2,
    save_out_jfn = 3,
    save_from_part = 4,
    save_to_part = 5,
    eoc = 6,

    num_of_functs = 8,      ! Number of functions parsed
                            ! in this example
    num_of_states = 11,     ! Number of command states
    fdb_size = 6,           ! Function data block size
                            ! plus one
    csb_size = 10,          ! Command state block size
    user = 5;               ! User defined word in fdb


FIELD state_fields =
    SET
    next = [18,18,0],
    action = [0,18,0]
    TES;

OWN
    state_table : BLOCKVECTOR[num_of_states, num_of_functs]
                        FIELD(state_fields) PRESET(

    [s0,cmini,next] = s1,    [s0,cmini,action] = null,
    [s1,cmswi,next] = s3,    [s1,cmswi,action] = null,
    [s1,cmifi,next] = s2,    [s1,cmifi,action] = save_input_info,
    [s2,cmswi,next] = s4,    [s2,cmswi,action] = null,
    [s2,cmqst,next] = s8,    [s2,cmqst,action] = save_from_part,
    [s3,cmofi,next] = s5,    [s3,cmofi,action] = save_out_jfn,
    [s4,cmofi,next] = s6,    [s4,cmofi,action] = save_out_jfn,
    [s5,cmifi,next] = s6,    [s5,cmifi,action] = save_in_jfn,
    [s6,cmnoi_from,next]=s7,[s6,cmnoi_from,action] = null,
    [s7,cmqst,next] = s8,    [s7,cmqst,action] = save_from_part,
    [s8,cmnoi_to,next]=s9,   [s8,cmnoi_to,action] = null,
    [s9,cmqst,next] = s10,   [s9,cmqst,action] = save_to_part,
    [s10,cmcfm,next] = s0,   [s10,cmcfm,action] = eoc);
```

```
LITERAL
    fw = -1;                                            (4)

OWN                                                     (5)
    fdb_ini : monblock[fdb_size] PRESET (
        [$cmfnp,cm_fnc] = $cmini,
        [user,fw] = cmini
        ),
    fdb_eol : monblock[fdb_size] PRESET (
        [$cmfnp,cm_fnc] = $cmcfm,
        [user,fw] = cmcfm
        ),
    fdb_switch : monblock[fdb_size] PRESET (
        [$cmfnp,cm_fnc] = $cmswi,
        [$cmdat,fw] = swblk,
        [user,fw] = cmswi
        ),
    fdb_ifi : monblock[fdb_size] PRESET (
        [$cmfnp,cm_fnc] = $cmifi,
        [user,fw] = cmifi
        ),
    fdb_ofi : monblock[fdb_size] PRESET (
        [$cmfnp,cm_fnc] = $cmofi,
        [user,fw] = cmofi
        ),
    fdb_from : monblock[fdb_size] PRESET (
        [$cmfnp,cm_fnc] = $cmnoi,
        [$cmdat,fw] = CH$PTR(UPLIT(%ASCIZ'FROM')),
        [user,fw] = cmnoi_from
        ),
    fdb_to : monblock[fdb_size] PRESET (
        [$cmfnp,cm_fnc] = $cmnoi,
        [$cmdat,fw] = CH$PTR(UPLIT(%ASCIZ'TO')),
        [user,fw] = cmnoi_to
        ),
    fdb_quostr : monblock[fdb_size] PRESET (
        [$cmfnp,cm_fnc] = $cmqst,
        [user,fw] = cmqst
        );

BIND
    state0 = PLIT (fdb_ini),
    state1 = PLIT (fdb_ifi, fdb_switch),
    state2 = PLIT (fdb_switch, fdb_quostr),
    state3 = PLIT (fdb_ofi),
    state4 = state3,
    state5 = PLIT (fdb_ifi),
    state6 = PLIT (fdb_from),
    state7 = PLIT (fdb_quostr),
    state8 = PLIT (fdb_to),
    state9 = state7,
    state10 = PLIT (fdb_eol);

OWN
    state_address : VECTOR [num_of_states] INITIAL (state0,
state1,
        state2, state3, state4, state5, state6,
                state7,state8,state9,state10);
```

```
OWN
    csl : monblock[csb_size] PRESET (
        [$cmioj,fw] = $priin ^ 18 OR $priou,
        [$cmrty,fw] = CH$PTR(UPLIT(%ASCIZ'TR>')),
        [$cmbfp,fw] = CH$PTR(command_buffer),
        [$cmptr,fw] = CH$PTR(command_buffer),
        [$cmcnt,fw] = buffer_length,
        [$cmabp,fw] = CH$PTR(atom_buffer),
        [$cmabc,fw] = buffer_length,
        [$cmgjb,fw] = cmdgjb
        );

MACRO chars(num)[] =                          (6)
    %COUNT
    %IF %COUNT LSS num %THEN , chars(num) %FI %;

BIND
    transtbl = UPLIT (%CHAR (chars(127)) );

FORWARD ROUTINE

    CMDINIT : NOVALUE,          ! Initialization routine
    LEXGET,                     ! Command line parsing
    OPENFILES : NOVALUE,        ! Open input and output files
    BUILDTBL : NOVALUE,         ! Generate the character
                                ! translation table
    EXPTBL,                     ! Translate dashes into
                                ! characters (a-z)
    CHRVAL,                     ! Character check (numeric,
                                ! alphabetic...)
    FILIO : NOVALUE;            ! Reads and writes bytes from
                                ! and to a file
```

1. S0 to S10 are the symbolic names for the states used by the command state table.

2. CMINI to CMNOI are the function codes used as indexes to the command state table.

3. NULL to EOC are symbolic names given to the actions performed during command parsing.

4. FW contains a fullword mask.

5. FDB_INI to FDB_QUOSTR are function descriptor blocks used by the COMND JSYS to parse the command line. The user field was added to more easily access the state table entries.

6. Generate a table that contains the set of ASCII characters.

10.2.2 Routine TRANSMAIN

```
ROUTINE TRANSMAIN : NOVALUE =
!+
! FUNCTIONAL DESCRIPTION:
!
!    This is the main routine which controls the
!    flow of the program.
!-
```

```
BEGIN

DO                                          (1)
    BEGIN

    CMDINIT ();                             (2)

    WHILE true DO                           (3)

        IF NOT LEXGET() THEN EXITLOOP;
    END
UNTIL .eoc_seen;

OPENFILES ();                               (4)
BUILDTBL ();                                (5)
FILIO ();                                   (6)
END;
```

1. Read until the end of the command line.
2. Initialize necessary flags.
3. Read lexemes until either an error or the end of the command occurs.
4. Open the input and output files.
5. Complete the building of the translation table.
6. Perform the file copy and close the files.

10.2.3 Routine CMDINIT

```
ROUTINE CMDINIT : NOVALUE =
!+
! FUNCTIONAL DESCRIPTION:
!
!     Initialize the flags for command parsing
!     and initialize the current process state.
!
!-
    BEGIN

        eoc_seen = false;
        outjfn = -1;

        IF .reparse
        THEN
            BEGIN

                reparse = false;
                state = s1;
            END
        ELSE
            state = s0;

        reset ();
    END;
```

10.2.4 Routine LEXGET

```
ROUTINE LEXGET =
!+
! FUNCTIONAL DESCRIPTION:
!
!     This routine does the command line parsing
!     using the state table in Section 10.2.1.
!
! FORMAL PARAMETERS:
!
!       None
!
! IMPLICIT INPUTS:
!
!      The routine assumes that STATE has been set up
!      with the appropriate number.
!
! IMPLICIT OUTPUTS:
!
!       The following locations may be modified:
!       STATE, INJFN, OUTJFN, EOC_SEEN, REPARSE,
!       TO_BUFFER, FROM_BUFFER, FILE_SPEC
!
! ROUTINE VALUE:
!
!       True - if a valid field is parsed
!       False - if an error, reparse, or an end of
!               command is seen
!
!-
    BEGIN

    LOCAL
        oldstate,
        fdb : REF monblock[fdb_size],                (1)
        first_fdb : REF monblock[fdb_size],          (2)
        prev_fdb : REF monblock[fdb_size],           (3)
        flgs :  MONWORD,
        fld_data,
        fd;

    BIND
        valid_token = .state_address [.state] : VECTOR;

    first_fdb = 0;

    INCR i FROM 0 TO .valid_token [-1] - 1 DO        (4)
        BEGIN

        IF .first_fdb EQL 0
        THEN
            first_fdb = .valid_token [.i]
        ELSE
            prev_fdb [$cmfnp,cm_lst] =
                      .valid_token [.i];

        prev_fdb = .valid_token [.i];
        END;

    prev_fdb [$cmfnp,cm_lst] = 0;
    comnd (csl, .first_fdb; flgs, fld_data, fd);     (5)
```

```
IF .flgs[cm_nop]                                    (6)
THEN
    BEGIN
    erstr ($priou, ($fhslf^18) OR .fld_data, 0);
    RETURN false
    END;

IF .flgs[cm_rpt]                                    (7)
THEN
    BEGIN
    reparse = true;
    RETURN false
    END;
```

1. FDB is the function descriptor block.

2. FIRST_FDB is the first function descriptor block.

3. PREV_FDB is the previous function descriptor block.

4. Set up a linked list of valid function descriptor blocks according to the state being processed.

5. Parse a field in the command.

6. Return an error if an unexpected field is detected.

7. Check for reparsing.

```
fdb = .fd<0, 18>;                                   (1)

oldstate = .state;
state = .state_table[.oldstate,.fdb[user,fw],next];

CASE .state_table[.oldstate,.fdb[user,fw],action]
    FROM 0 TO eoc OF
    SET

    [null] :
        BEGIN

        RETURN true
        END;

    [save_input_info] :
        BEGIN

        injfn = .fld_data;                          (2)

        JFNS(CH$PTR(file_spec),.injfn,              (3)
            FLD($JSAOF,JS_NAM)+FLD($JSAOF,JS_TYP)
                +JS_PAF,0);

        fdb_ofi[$cmdef,fw] = CH$PTR(file_spec);     (4)

        fdb_ofi[$cmfnp,cm_dpp] = true;              (5)
        RETURN true
        END;
```

```
    [save_in_jfn] :
        BEGIN

        injfn = .fld_data;                                  (6)
        RETURN true
        END;

    [save_out_jfn] :
        BEGIN

        outjfn = .fld_data;                                 (7)
        RETURN true
        END;

    [save_from_part] :                                      (8)
        BEGIN

        CH$MOVE (buffer_length, CH$PTR (atom_buffer),
                 CH$PTR (from_buffer));
        RETURN true
        END;

    [save_to_part] :                                        (9)
        BEGIN

        CH$MOVE (buffer_length, CH$PTR (atom_buffer),
                 CH$PTR (to_buffer));
        RETURN true
        END;

    [eoc] :                                                 (10)
        BEGIN

        eoc_seen = true;
        RETURN false
        END;

    [OUTRANGE] :                                            (11)
        BEGIN

        jsys_detected_error;
        RETURN false
        END;
    TES

END;
```

1. Compute the new state and perform appropriate actions based on the current state and the input token parsed by the COMND JSYS.

   Note that FBD now equals the function descriptor block that was used.

2. Save the input JFN.

3. Save the filename and the file type.

4. Default the output file spec to FDB_OFI.

5. Flag the COMND JSYS to indicate that a default has been found.

6. Save the input JFN.

7. Save the output JFN.

8. Save the (FROM) quoted string.

9. Save the (TO) quoted string.

10. Set the end of the command flag.

11. We should never get to an OUTRANGE.

10.2.5 Routine OPENFILES

```
ROUTINE OPENFILES : NOVALUE =
!+
! FUNCTIONAL DESCRIPTION:
!
!       Opens the input and output files. If an output
!       file was not specified, we get a JFN for a new
!       generation of the input file spec.
!
! FORMAL PARAMETERS:
!
!       None
!
! IMPLICIT INPUTS:
!
!       The INJFN must contain input file JFN.
!
!       The OUTJFN contains output file JFN or -1.
!
! IMPLICIT OUTPUTS:
!
!       The OUTFJN may be modified
!
! ROUTINE VALUE:
!
!       None
!-
    BEGIN

    IF NOT openf(.injfn,                                (1)
                 (fld (7, of_bsz) OR of_rd))
    THEN
        jsys_detected_error;

    IF .outjfn LSS 0                                    (2)
    THEN
        IF NOT gtjfn ((gj_sht OR gj_fou),
                      CH$PTR (file-spec); outjfn)
        THEN
            jsys_detected_error;

    IF NOT openf (.outjfn, (fld (7, of_bsz) OR of_wr))
    THEN
        jsys_detected_error;

    END;
```

1. Open the input file in 7-bit mode.

2. Check if an output file was specified.

10.2.6 Routine BUILDTBL

```
ROUTINE BUILDTBL : NOVALUE =
!+
! FUNCTIONAL DESCRIPTION:
!
!       This routine deals with the (FROM) and (TO) quoted
!       strings. The routine calls EXPTBL to replace the
!       abbreviated form of "a1-a2" with all the characters
!       between and checks their validity.
!
!       If the routine finds that the (TO) argument is empty
!       it fills the buffer with zeros to indicate that the
!       (FROM) characters are to be deleted.
!
!       If the number of characters in the (FROM) part exceeds
!       the number of characters in the (TO) part then the last
!       character in the (TO) part is replicated until the (TO)
!       string is as long as the (FROM) string.
!
!       Finally, the translation table used for file I/O will be
!       updated to reflect the specified character translations.
!
!
! FORMAL PARAMETERS:
!
!       None
!
! IMPLICIT INPUTS:
!
!       FROM_BUFFER, TO_BUFFER, FROM_COUNT, TO_COUNT
!
! IMPLICIT OUTPUTS:
!
!       The following locations are modified:
!
!       FROM_COUNT,TO_COUNT,FROM_BUFFER,TO_BUFFER, TRANSTBL
!
! ROUTINE VALUE:
!
!       None
!-
    BEGIN

    LOCAL
        chr_ptr;

    from_count = exptbl (from_buffer);            (1)
    to_count = exptbl (to_buffer);

    IF (.from_count LSS .to_count) OR (.from_count EQL 0)
    THEN
        quote_error;

    IF .to_count EQL 0                             (2)
    THEN
        CH$FILL (0, .from_count, CH$PTR (to_buffer))
    ELSE

        IF .to_count NEQ .from_count
        THEN
            BEGIN
```

```
            chr_ptr = CH$PTR (to_buffer, .to_count - 1);

            CH$FILL (CH$RCHAR_A (chr_ptr),
                     .from_count - .to_count,
                     .chr_ptr);
            END;

        chr_ptr = CH$PTR (from_buffer);                (3)

        INCR i FROM 0 TO .from_count - 1 DO
            BEGIN

            LOCAL
                char;
                char = CH$RCHAR_A (chr_ptr);
                CH$WCHAR (CH$RCHAR (CH$PTR (to_buffer, .i)),
                          CH$PTR (transtbl, .char));
            END

        END;
```

1. Expand the buffers to include all "a1-a2" characters, and save the number of characters.

2. A (TO) count of zero indicates characters are to be deleted.

3. A (TO) count that is less than a (FROM) count indicates that the last (TO) character will be replicated.

4. The translation table is modified so that each character to be translated is replaced by the translation.

10.2.7 Routine EXTBL

```
ROUTINE  EXPTBL (buffer) =
!+
! FUNCTIONAL DESCRIPTION:
!
!       Reads the characters from the given buffer
!       and, if the 'a1-a2' form is used, fills in
!       in appropriate characters.
!
! FORMAL PARAMETERS:
!
!       buffer - address of characters to be examined
!
! IMPLICIT INPUTS:
!
!       None
!
! IMPLICIT OUTPUTS:
!
!       Modifies BUFFER,ATOM_BUFFER
!
! ROUTINE VALUE:
!
!       Number of characters in the buffer
!-
```

```
BEGIN

LOCAL
    char,
    prev_char,
    dst_ptr,
    chr_ptr,
    chr_cnt,
    temp_buffer :  VECTOR[CH$ALLOCATION(buffer_length)];

chr_cnt = prev_char = 0;
dst_ptr = CH$PTR (temp_buffer);
chr_ptr = CH$PTR (.buffer);
char = CH$RCHAR_A (chr_ptr);                    (1)

WHILE .char NEQ 0 DO                            (2)
    BEGIN

    IF .char EQL %C'-'                          (3)
    THEN
        BEGIN

        IF .prev_char EQL 0 THEN quote_error; (4)

        char = CH$RCHAR_A (chr_ptr);

        IF (.char LEQ .prev_char)               (5)
             OR (chrval (.prev_char) NEQ chrval (.char))
        THEN
            quote_error;

        INCR i FROM .prev_char + 1 TO .char DO  (6)

            BEGIN
            chr_cnt = .chr_cnt + 1;

            IF .chr_cnt GTR buffer_length
            THEN
                quote_error;

            CH$WCHAR_A (.i, dst_ptr);
            END;

        prev_char = 0;
        END
    ELSE
        BEGIN

        IF chrval (.char) EQL not_alpha_numeric
        THEN
            quote_error:

        IF (chr_cnt = .chr_cnt + 1) GTR buffer_length
        THEN
            quote_error;

        CH$WCHAR_A (.char, dst_ptr);
        prev_char = .char;
        END;

    char = CH$RCHAR_A (chr_ptr);
    END;

CH$MOVE (.chr_cnt, CH$PTR (temp_buffer), CH$PTR (.buffer));
RETURN .chr_cnt
END;
```

1. Read the first character.

2. Read all characters.

3. Check for the form 'a1-a2'.

4. Do not allow "-z" or "a--z".

5. Check the validity of the range.

6. Include in the buffer all characters that are within range.

10.2.8 Routine CHRVAL

```
ROUTINE CHRVAL (chr) =
!+
! FUNCTIONAL DESCRIPTION:
!
!       This routine checks whether a given character is a
!       number, an upper_case letter, or a lower_case letter.
!
! FORMAL PARAMETERS:
!
!       chr - Character to check
!
! IMPLICIT INPUTS:
!
!       None
!
! IMPLICIT OUTPUTS:
!
!       None
!
! ROUTINE VALUE:
!
!       numeric - Character is a number
!       alpha - Character is an upper_case letter
!       alpha_lower_case - Character is a lower_case letter
!       not_alpha_numeric - Character is not a letter or a digit
!-
    BEGIN

    SELECTONE .chr OF
        SET

        [%C'0' TO %C'9'] :
            RETURN numeric;

        [%C'A' TO %C'Z'] :
            RETURN alpha;

        [%C'a' TO %C'z'] :
            RETURN alpha_lower_case;

        [OTHERWISE] :
            RETURN not_alpha_numeric;
        TES;

    END;
```

10.2.9 Routine FILIO

```
ROUTINE FILIO : NOVALUE =
!+
! FUNCTIONAL DESCRIPTION:
!
!       This routine reads each character of the input file to
!       the output file, along with the corresponding translated
!       character. A zero in the translation table indicates that
!       the character is to be omitted from the output file. The
!       routine then closes both the input and output file.
!
! FORMAL PARAMETERS:
!
!       None
!
! IMPLICIT INPUTS:
!
!       OUTJFN, INJFN, TRANSTBL all set up
!
! IMPLICIT OUTPUTS:
!
!       None
!
! ROUTINE VALUE:
!
!       None
!
!-
    BEGIN

    LOCAL
        char,
        cond : monword,
        eof;

    WHILE NOT .eof DO
        BEGIN

        IF NOT bin(.injfn; char)        (1)
        THEN
            BEGIN
            GTSTS(.injfn;cond);         (2)
            IF .cond[GS_EOF]
            THEN
                eof=true                (3)
            ELSE
                jsys_detected_error     (4)
            END;

        IF .char NEQ 0                  (5)
        THEN
            BEGIN

            char = CH$RCHAR (CH$PTR (transtbl, .char));
            IF .char NEQ 0 THEN bout (.outjfn, .char);
            END
        ELSE
            bout (.outjfn, .char);
        END;
```

```
    closf (.injfn);
    closf (.outjfn);
    END;
END

ELUDOM
```

1. Read a character from the input file.

2. Check the error conditions.

3. End of file is found.

4. Report any other type of error.

5. If the input character is not a null, and will not translate to a null character, the character is written to the output file.

APPENDIX A

SUMMARY OF COMMAND SYNTAX

This appendix summarizes the command syntax and switch defaults for TOPS-20 and TOPS-10.

A.1 TOPS-20 COMMAND SUMMARY

bliss-compilation	BLISS>bliss-command-line ...
bliss-command-line	{ switch ... } space input-spec ,...
input-spec	file-spec+... { switch ... }
space	blank ...
switch	(output-switch general-switch check-switch terminal-switch optimization-switch listing-switch reference-switch environment-switch)
output-switch	(/OBJECT {:file-spec} | /NOOBJECT) (/LISTING {:file-spec} | /NOLISTING) (/LIBRARY {:file-spec} | /NOLIBRARY) (/MASTER-CROSS-REFERENCE {:file-spec} | /NOMASTER-CROSS-REFERENCE)
general-switch	(/DEBUG | /NODEBUG) (/CODE | /NOCODE) (/VARIANT {:n}) (/ERROR-LIMIT {:n})
check-switch	/CHECK: { check-value } { (check-value,...) }
check-value	(FIELD | NOFIELD) (INITIAL | NOINITIAL) (OPTIMIZE | NOOPTIMIZE) (REDECLARE | NOREDECLARE)

```
terminal-switch          { /ERRS        |  /NOERRS        }
                         { /STATISTICS  |  /NOSTATISTICS  }

optimization-switch      ( /OPTLEVEL : { 0 | 1 | 2 | 3 } )
                         ( /OPTIMIZE  |  /NOOPTIMIZE     )
                         ( /SAFE      |  /NOSAFE         )
                         ( /ZIP       |  /NOZIP          )
                         ( /QUICK     |  /NOQUICK        )

listing-switch           ( /PAGSIZ: { 20 | 21 | 22 |... } )
                         ( /HEADER  |  /NOHEADER          )
                         ( /UNAMES  |  /NOUNAMES          )
                         ( /FORMAT  : { option       }    )
                         (            {(option,...)  }    )

option                   ( ASSEMBLY     |  NOASSEMBLY     )
                         ( BINARY       |  NOBINARY       )
                         ( COMMENTARY   |  NOCOMMENTARY   )
                         ( EXPAND       |  NOEXPAND       )
                         ( LIBRARY      |  NOLIBRARY      )
                         ( OBJECT       |  NOOBJECT       )
                         ( REQUIRE      |  NOREQUIRE      )
                         ( SOURCE       |  NOSOURCE       )
                         ( SYMBOLIC     |  NOSYMBOLIC     )
                         ( TRACE        |  NOTRACE        )

reference-switch         { /CROSS-REFERENCE       { :{ reference-value       } } }
                         { /MASTER-CROSS-REFERENCE    :{ (reference-value,...) } } }
                                                      { :file-spec }

reference-value          { MULTIPLE | NOMULTIPLE }

environment-switch       ( /KA10 | /KI10 | /KL10 | /KS10 )
                         ( /TOPS10 | /TOPS20             )
                         ( /EXTENDED | /NOEXTENDED       )
                         ( /EXTENDED : SECTION-INDEPENDENT )
```

A.2 TOPS-20 SWITCH DEFAULTS

The following are the switch defaults for the TOPS-20:

/OBJECT:input-file-name.REL

/NOLISTING

/NOLIBRARY

/NODEBUG

/CHECK:(FIELD,INITIAL,OPTIMIZE,NOREDECLARE)

/CODE

/NOCROSS-REFERENCE or /CROSS-REFERENCE(NOMULTIPLE)

```
/VARIANT:0

/ERRS

/ERROR-LIMIT:30

/NOMASTER-CROSS-REFERENCE

/NOSTATISTICS

/OPTLEVEL:2

/OPTIMIZE

/SAFE

/NOZIP

/NOQUICK

/PAGSIZ:52

/HEADER

/NOUNAMES

/FORMAT:(NOASSEMBLY, BINARY, COMMENTARY, NOEXPAND, NOLIBRARY,
         OBJECT, NOREQUIRE, SOURCE, SYMBOLIC, NOTRACE)

/KL10

/TOPS20

/NOEXTENDED
```

A.3 TOPS-10 COMMAND SUMMARY

bliss-command-line	{output-file-list} = source-file-list {switch...}
source-file-list	source-file-spec,...
output-file-list	{ object-file-spec ,... ,listing-file-spec ,... ,master-cref-spec }
object-file-spec listing-file-spec source-file-spec master-cref-spec	} file-spec

file-spec	{ device: } file-name { .file-type } { [ppn] }
device	any logical or physical device name
file-name	1 to 6 alphanumeric characters
file-type	0 to 3 alphanumeric characters
ppn	project-number,programmer-number

```
switch                  ( library-switch
                        | general-switch
                        | check-switch
                        | terminal-switch
                        | optimization-switch
                        | source-list-switch
                        | reference-switch
                        | environment-switch )

library-switch          { /LIBRARY  |  /NOLIBRARY }

general-switch          ( /DEBUG | /NODEBUG )
                        ( /CODE  | /NOCODE  )
                        ( /VARIANT {:n}     )
                        ( /ERRLIM {:n}      )

check-switch            /CHECK : { check-value          }
                                 { (check-value ,...)   }

check-value             ( FIELD      | NOFIELD      )
                        ( INITIAL    | NOINITIAL    )
                        ( OPTIMIZE   | NOOPTIMIZE   )
                        ( REDECLARE  | NOREDECLARE  )

terminal-switch         { /ERRS        |  /NOERRS       }
                        { /STATISTICS  |  /NOSTATISTICS }

optimization-switch     ( /OPTLEVEL : { 0 | 1 | 2 | 3 } )
                        ( /SAFE       |  /NOSAFE       )
                        ( /ZIP        |  /NOZIP        )
                        ( /QUICK      |  /NOQUICK      )

source-list-switch      ( /PAGSIZ: { 20 | 21 | 22 |... } )
                        ( /HEADER  |  /NOHEADER          )
                        ( /UNAMES  |  /NOUNAMES          )
                        ( /LIST    : { option       }    )
                                     {(option,... )}     )

option                  ( ASSEMBLY    | NOASSEMBLY    )
                        ( BINARY      | NOBINARY      )
                        ( COMMENTARY  | NOCOMMENTARY  )
                        ( EXPAND      | NOEXPAND      )
                        ( LIBRARY     | NOLIBRARY     )
                        ( OBJECT      | NOOBJECT      )
                        ( REQUIRE     | NOREQUIRE     )
                        ( SOURCE      | NOSOURCE      )
                        ( SYMBOLIC    | NOSYMBOLIC    )
                        ( TRACE       | NOTRACE       )

reference-switch        /CREF { { reference-value        } }
                              {:{ (reference-value ,...) } }

reference-value         { MULTIPLE | NOMULTIPLE }

environment-            { /KA10 | /KI10 | /KL10 | /KS10 }
 switch                 { /TOPS10 | /TOPS20             }
```

A.4 TOPS-10 SWITCH DEFAULTS

The following are the switch defaults for the TOPS-10:

```
/NOLIBRARY

/NODEBUG

/CHECK:(FIELD,INITIAL,OPTIMIZE,NOREDECLARE)

/CODE

/NOCREF or /CREF(NOMULTIPLE)

/VARIANT:0

/ERRS

/ERRLIM:30

/NOSTATISTICS

/OPTLEVEL:2

/SAFE

/NOZIP

/NOQUICK

/PAGSIZ:52

/HEADER

/NOUNAMES

/LIST:(NOASSEMBLY, BINARY, COMMENTARY, NOEXPAND, NOLIBRARY,
       OBJECT, NOREQUIRE, SOURCE, SYMBOLIC, NOTRACE)

/KA10

/TOPS10
```

APPENDIX 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 are indented to the same level as the BEGIN-END delimiters.

The format for a declaration is:

```
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 appear on separate lines from subparts and subparts are indented one tab. For example, IF expressions are written in either of two formats:

```
IF test THEN consequence ELSE alternative;
```

or

```
IF test
THEN
    consequence
ELSE
    alternative;
```

The examples used in Chapter 3 are indented correctly, although all comments have been omitted in order to save space.

APPENDIX 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. 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 may 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 with VAX/VMS, and on BLI: with TOPS-10 and TOPS-20.

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
```

APPENDIX D

IMPLEMENTATION LIMITS

Each BLISS-36 compiler implementation has 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-36 LANGUAGE

The maximum number of	is
• 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
• Words initialized by a single PLIT (that is, the maximum word count of a single PLIT)	262,143

D.2 SYSTEM INTERFACES

The maximum number of	is
• Characters in an input source line	131
• Simultaneously active (depth of nested) REQUIRE files	9

APPENDIX 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 close 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 comma(s), 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 semi-colon 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 comma(s) 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 non-printing ASCII characters has been used (as other than data) in a BLISS module.

User Action: Only four non-printing characters (blank, tab, vertical-tab, and form-feed) may be used in coding.

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: n 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- or 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 value of the low-bound range of a case-expression must not exceed the value of the high-bound range.

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 comma(s) 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 exitloop-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(s) (END, ")", "]", ">") or add the missing left bracket(s) (BEGIN, "(", "[", "<").

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 coding of the module-switches or switches-declaration.

User Action: Correct the coding of 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 attribute(s) 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 comma(s) 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 parenthesis must be used to replace a "(" -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 declaration coding.

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.

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 comma(s) 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 comma(s) 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 coding of structure-reference.

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 comma(s) 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 missing comma(s) 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 coding of 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 semicolon(s) 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: Builtin 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-ltce declaration should not appear within the outer most level of the module-body.

User Action: Correct the use of expressions and declarations within the outer most 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) simultaneously active.

User Action: Redistribute explicit register usage to prevent overlaps as regards 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 named 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 attribute(s).

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 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 imbedded 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 terminator(s).

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 labelled expression must be contained within a BEGIN-END or parenthesis pair.

User Action: Enclose the expression(s) 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 parenthesis.

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 outer most 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 coding.

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 comma(s) 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 builtin-function must not exceed the number of formal-parameters used in the builtin-routine.

User Action: Correct actual-parameter usage in call to builtin-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 builtin-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 builtin-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 comment(s) 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 comma(s) 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 three.

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 outer most 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 comma(s) 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 been illegally coded.

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 builtin.

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 are coded 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 coding of 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.

168 Illegal value <value> in LINKAGE declaration

Explanation: A literal value exceeds the permitted range.

User Action: Typically skips (N) value, where N is -1 to 2.

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 sign(s).

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 comma(s).

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 bracket(s).

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 a TES.

User Action: Insert missing comma(s) 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 bracket(s).

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 comma(s) and/or the close parenthesis.

185 Invalid linkage attribute in LINKAGE declaration

Explanation: A linkage-option has been used in a linkage-declaration that is invalid.

User Action: Use a valid modifier in the linkage-declaration.

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 at 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 via 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 hat 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 declaration coding 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-spec 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 may be used in a library source file.

User Action: Remove the invalid declaration(s) 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 builtin 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 builtin-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 builtin-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, which is: oooa(oooooo)-o; where "o" is an octal digit and "a" is an alphabetic.

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 Expression(s) 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 dialect(s) 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 dialect(s) defined by the language switch.

User Action: Evaluate the feature named and take appropriate action.

216 Language feature not transportable:

Explanation: The feature specified by the keyword shown is not transportable to the dialect(s) 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 builtin when a FORTRAN linkage-routine is being compiled.

User Action: Add an explicit builtin-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 coding of 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: Assure 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 builtin-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 builtin-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 OTS_LINKAGE 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 OTS_LINKAGE 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 OTS_LINKAGE 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 OTS_LINKAGE 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 Reference to uninitialized LOCAL, STACKLOCAL, or REGISTER symbol <name>

Explanation: The routine contains a reference to the value of the variable named prior to its first assignment. This message can also occur if a reference and an assignment to the same variable occurs in different branches of an IF, CASE, or SELECT statement contained within a loop.

User Action: Intitialize the symbol prior to its first use.

251 Symbol <name> is declared <class> in an outer block

Explanation: A symbol name declared in an inner block is inaccessible because it is also declared in an outer 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 actions statements in a SELECT or SELECTONE construct cannot be compiled and consequently certain actions have been eliminated.

User Action: Evaluate the select statements for proper operation.

E.1 BLISS COMPILER FATAL ERRORS

The following fatal error messages indicate serious problems with the environment and/or compiler. 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 and -20, the user's program may be too large to compile.

I/O ERROR ON INPUT FILE

An error occurred while accessing an input file. This may be preceded by other error messages which provide more specific information about the error.

I/O ERROR ON OBJECT FILE

An error occurred while accessing the output object file. This may be preceded by other error messages which provide more specific information about the error.

I/O ERROR ON LISTING FILE

An error occurred while accessing the output listing file. This may be preceded by other error messages which provide more specific information about the error.

I/O ERROR ON LIBRARY FILE

An error occurred while accessing a BLISS precompiled library file. This may be preceded by other error messages which provide more specific information about the error.

LIBRARY PRE-COMPILATION EXCEEDS COMPILER LIMIT

A pre-compiled BLISS library cannot be larger than approximately 2048 (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 VAX-VMS only, for BLISS-32 or BLISS-16. The user's 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 parsing the command line.

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 VAX-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.

APPENDIX F

SAMPLE OUTPUT LISTING

The following pages contain the complete output listing for the module TESTFACT. Chapter 3 examples use excerpts from this listing.

```
TESTFACT                                        3-Jun-1983 18:24:36    TOPS-20 Bliss-36 3A(200)     Page   1
                                                2-May-1983 15:27:31    PS:<DIRECTORY>MYPROG.B36.6 (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
        TWOSEG


        .REQUEST  SYS:B362LB.REL


        RELOC   0                               ;                       000000'
A:      BLOCK   1                               ;                       000000'
B:      BLOCK   1                               ;                       000001'


        EXTERN  REULT


AC0=    0
AC1=    1
AC2=    2
AC3=    3
AC4=    4
AC5=    5
AC6=    6
AC7=    7
AC10=   10
AC11=   11
AC12=   12
AC13=   13
AC14=   14
FP=     15
AC16=   16
```

Figure F-1: Sample Output Listing

```
TESTFACT                                        3-Jun-1983 18:24:36    TOPS-20 Bliss-36 3A(200)     Page   2
                                                2-May-1983 15:27:31    PS:<DIRECTORY>MYPROG.B36.6 (1)

SP=     17


        RELOC   400000                  ;                               400000'
IFACT:  MOVEI   AC1,1                   ; RESULT,1                      400000' 201 01 0 00 000001  0012
        MOVEI   AC2,1                   ; I,1                           400001' 201 02 0 00 000001  0013
        JRST    L.2                     ; L.2                           400002' 254 00 0 00 400005'
L.1:    MOVE    AC1,REULT               ; RESULT,REULT                  400003' 200 01 0 00 000000*  0014
        IMUL    AC1,AC2                 ; RESULT,I                      400004' 220 01 0 00 000002
L.2:    ADDI    AC2,1                   ; I,1                           400005' 271 02 0 00 000001  0013
        CAMG    AC2,-1(SP)              ; I,N                           400006' 317 02 0 17 777777
        JRST    L.1                     ; L.1                           400007' 254 00 0 00 400003'
        POPJ    SP,                     ; SP,                           400010' 263 17 0 00 000000  0008

; Routine Size:  9 words


;   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:  PUSH    SP,AC16                 ; SP,AC16                       400011' 261 17 0 00 000016  0018
        MOVE    AC16,-2(SP)             ; AC16,N                        400012' 200 16 0 17 777776  0019
        CAIG    AC16,1                  ; AC16,1                        400013' 307 16 0 00 000001
        JRST    L.3                     ; L.3                           400014' 254 00 0 00 400024'
        MOVE    AC1,AC16                ; AC1,AC16                      400015' 200 01 0 00 000016  0021
        SUBI    AC1,1                   ; AC1,1                         400016' 275 01 0 00 000001
        PUSH    SP,AC1                  ; SP,AC1                        400017' 261 17 0 00 000001
        PUSHJ   SP,RFACT                ; SP,RFACT                      400020' 260 17 0 00 400011'
        IMUL    AC1,AC16                ; AC1,AC16                      400021' 220 01 0 00 000016
        ADJSP   SP,-1                   ; SP,-1                         400022' 105 17 0 00 777777
        JRST    L.4                     ; L.4                           400023' 254 00 0 00 400025'  0019
L.3:    MOVEI   AC1,1                   ; AC1,1                         400024' 201 01 0 00 000001
L.4:    POP     SP,AC16                 ; SP,AC16                       400025' 262 17 0 00 000016  0018
        POPJ    SP,                     ; SP,                           400026' 263 17 0 00 000000

; Routine Size:  14 words


;   0024  1
;   0025  1     ROUTINE MAINPROG :NOVALUE =
;   0026  2         BEGIN
;   0027  2         A = IFACT (5);
;   0028  2         B = RFACT (5);
;   0029  1         END;
```

Figure F-1 (Cont.): Sample Output Listing

```
TESTFACT                                        3-Jun-1983 18:24:36    TOPS-20 Bliss-36 3A(200)     Page   3
                                                2-May-1983 15:27:31    PS:<DIRECTORY>MYPROG.B36.6 (1)

MAINPROG:
        PUSH    SP,C.1                  ; SP,[5]                        400027' 261 17 0 00 400037'  0027
        PUSHJ   SP,IFACT                ; SP,IFACT                      400030' 260 17 0 00 400000'
        MOVEM   AC1,A                   ; AC1,A                         400031' 202 01 0 00 000000'
        PUSH    SP,C.1                  ; SP,[5]                        400032' 261 17 0 00 400037'  0028
        PUSHJ   SP,RFACT                ; SP,RFACT                      400033' 260 17 0 00 400011'
        MOVEM   AC1,B                   ; AC1,B                         400034' 202 01 0 00 000001'
        ADJSP   SP,-2                   ; SP,-2                         400035' 105 17 0 00 777776  0026
        POPJ    SP,                     ; SP,                           400036' 263 17 0 00 000000  0025
C.1:    EXP     5                       ; 5                             400037' 000000  000005

; Routine Size:  9 words


;   0030  1
;   0031  1     END
;   0032  0     ELUDOM


        RELOC   2                       ;                               000002'
.STACK.:BLOCK   4000                    ;                               000002'


        RELOC   400040                  ;                               400040'
.MAIN.: TDZA    AC1,AC1                 ; AC1,AC1                       400040' 634 01 0 00 000001  0000
        MOVEI   AC1,1                   ; AC1,1                         400041' 201 01 0 00 000001
        JSYS    147                     ; 147                           400042' 104 00 0 00 000147
        MOVE    AC2,C.2                 ; AC2,[-4000,,.STACK.-1]        400043' 200 02 0 00 400057'
        PUSH    AC2,SP                  ; AC2,SP                        400044' 261 02 0 00 000017
        PUSH    AC2,AC11                ; AC2,AC11                      400045' 261 02 0 00 000011
        PUSH    AC2,AC7                 ; AC2,AC7                       400046' 261 02 0 00 000007
        PUSH    AC2,AC0                 ; AC2,AC0                       400047' 261 02 0 00 000000
        PUSH    AC2,AC1                 ; AC2,AC1                       400050' 261 02 0 00 000001
        MOVE    SP,AC2                  ; SP,AC2                        400051' 200 17 0 00 000002
        SETZB   FP,EFPNT.               ; FP,EFPNT.                     400052' 403 15 0 00 000000*
        PUSHJ   SP,MAINPROG             ; SP,MAINPROG                   400053' 260 17 0 00 400027'
        ADJSP   SP,-5                   ; SP,-5                         400054' 105 17 0 00 777773
L.5:    JSYS    170                     ; 170                           400055' 104 00 0 00 000170
        JRST    L.5                     ; L.5                           400056' 254 00 0 00 400055'
C.2:    XWD     -4000,.STACK.-1         ; -4000,.STACK.-1               400057' 774000  000001'

; Routine Size:  16 words

        END     .MAIN.

; Low  segment length:    2050 words
; High segment length:      48 words



; Information:  0
```

Figure F-1 (Cont.): Sample Output Listing

```
; Warnings:      2
; Errors:        0

; Size:          48 code + 2050 data words
; Run Time:         00:00.8
; Elapsed Time:     00:03.7
; Lines/CPU Min:     2382
; Lexemes/CPU-Min:  9528
; Memory Used:  3 pages
; Compilation Complete
```

Figure F-1 (Cont.): Sample Output Listing

APPENDIX G

MIXING BLISS-36 MODULES AND BLISS-10 MODULES

The need may arise for a programmer to have both BLISS-36 and BLISS-10 modules in a single program: for example, for an incremental conversion of a BLISS-10 program to BLISS-36. (Refer to CVT10 in Chapter 9.) Another example would be for writing a BLISS-36 program that uses a package written in BLISS-10 or an assembly language package that was designed for use by BLISS-10 modules.

Both BLISS-36 and BLISS-10 can conveniently generate code that uses either the BLISS10 linkage (stack-pointer=0, frame-pointer=2, value-register=3) or the BLISS36C linkage (stack-pointer=15, frame-pointer=13, value-register=1).

To generate code that uses BLISS10 linkage:

- In BLISS-36 modules, use the module switches LINKAGE (BLISS10) and ENVIRONMENT (BLISS10_OTS).
- In BLISS-10 modules, the default is to generate code that uses BLISS10 linkage conventions.

To generate code that uses BLISS36C linkage:

- In BLISS-36 modules, the default linkage convention is BLISS36C.
- In BLISS-10 modules, use the /Z switch in the BLISS-10 command line, or use the following module switches:

```
SREG = 15, FREQ = 13, VREG = 1, DREGS = 7, RESERVE(0,14)
```

Code generated by BLISS-10 uses utility routines to save and restore registers. These are generated as part of a BLISS-10 module that contains the STACK or PROLOG module switch or is compiled with the /P command-line switch.

APPENDIX H

USER-GENERATED OTS FILES

The BLISS-36 object-time system (OTS) contains routines that implement various character handling functions and the condition handling mechanism. OTS object files use either BLISS-10 or BLISS-36C linkage conventions. The user determines which convention is to be used by setting either the BLISS10_OTS or BLISS36C_OTS (default) environment switch.

The BLISS-36 dialect of the BLISS language permits a programmer to define a linkage convention that is incompatible with either the BLISS-10 or BLISS-36C linkage. Incompatible means that a routine using one linkage could not call a routine that uses the other linkage. This would be the case if calling and called routines used different registers as stack pointers. A programmer may be forced to use such a linkage when interfacing to a set of utility routines written in assembly language or in BLISS-10 with non-default linkage registers.

The compiler generates in the object file a request to the linker for a user-generated OTS file and code calls to it using a user specified linkage when OTS and OTS_LINKAGE switches appear in the module header.

To generate an OTS file, create a file (e.g., LINKAGE.MAC) having the following format:

```
DEFINE TYPE, < . . . >
```

where the ellipsis represents one or more of the macros discussed later. Then, starting at monitor command level, type the following:

TOPS-20	TOPS-10
@MACRO	.R MACRO
*USROTS=LINKAGE,BLSOTS	*USROTS=LINKAGE,BLSOTS
*^C	*^C

The OTS source file (BLSOTS.MAC) is assumed to be somewhere in the user's search path. If it is not, an appropriate device should be defined and specified.

If no linkage file is included, the OTS is assembled with the BLISS-36C linkage.

The first line of TYPE should specify either PUSHJ or F10, which correspond to the BLISS-36 linkage types of the same names. These must be the first item in TYPE. If neither is specified, PUSHJ is assumed. LINKAGEREGS, PRESERVE, and NOPRESERVE declarations may then appear in any order. Each should be put on a separate line.

The LINKAGEREGS declaration may take from zero to four arguments enclosed in angle brackets (< >). The first argument represents the register number to be used for the stack pointer (default is 0). The second argument represents the register number for the frame pointer (default is 2). Using a negative number or the default value indicates that a preserved register is to be used as a frame pointer if one is needed and no frame list is to be kept. The third argument represents the register number for the returned value (default is 3). The fourth argument represents the register number to be used for the applied pointer (default is 14). This argument has meaning only if the linkage type is F10; it is ignored if the linkage type is PUSHJ.

PRESERVE identifies registers whose contents should be saved by any routine that uses them. Any register that was previously listed in a NOPRESERVE declaration or that was or will be specified in a LINKAGEREGS declaration is ignored. If no PRESERVE declaration is included, the default is:

```
PRESERVE <0,11,12,13,14,15> (if linkage type is PUSHJ)

or

PRESERVE <> (if linkage type is F10)
```

NOPRESERVE identifies "temporary" registers; that is, they may be used freely, but may be modified by making a call. Any register previously listed in a PRESERVE declaration or that was or will be specified in a LINKAGEREGS declaration is ignored. If no NOPRESERVE declaration is included, the default is:

```
NOPRESERVE <1,2,3,4,5,6,7,8,9,10> (if linkage type is PUSHJ)

or

NOPRESERVE <1,2,3,4,5,6,7,8,9,10,11,12,13> (if linkage type is
F10)
```

All register numbers are specified in decimal radix.

The user may also include the declaration SET%KL. This specifies that the OTS is to be run on a KL-10 processor (for example, a DECSYSTEM-20). If SET%KL is not specified, the generated code is for a KA-10 processor and may be run on any DECsystem-10 or DECSYSTEM-20.

The TOPS20 macro may be used to indicate that the code is destined for a DECSYSTEM-20. This will have a side-effect of doing a SET%KL.

Macros are available for three of the four predefined linkages for BLISS-36. They may be used instead of PUSHJ, F10, LINKAGEREGS, PRESERVE, and NOPRESERVE. The macros are:

```
BLS36C                  corresponding to BLISS36C
BLSS10                  corresponding to BLISS10
FRTFUNC                 corresponding to FORTRAN_FUNC
```

A linkage corresponding to FORTRAN_SUB may not at this time be defined because of the absence of preserved registers. An OTS with linkage FRTFUNC is safely callable from a BLISS routine with linkage FORTRAN_SUB.

The declaration SET%PSECT generates a PSECTED OTS as opposed to an OTS segmented by default. A PSECTED OTS is required for programs running with extended addressing.

The default PSECT names are: OWN, $GLOBAL$, $PLIT$, and $CODE$; however, you can change these names by redefining the following macros:

```
PSECTOWN      (defaults to .PSECT $OWN$)
PSECTGLOBAL   (defaults to .PSECT $GLOBAL$)
PSECTPLIT     (defaults to .PSECT $PLIT$)
PSECTCODE     (defaults to .PSECT $CODE$)
```

For example, the OWN and $CODE$ PSECT can be personalized to MYOWN and MYCODE, respectively, as follows:

```
DEFINE TYPE,
    <
     PUSHJ
     TOPS20
     SET%PSECT
     LINKAGEREGS<15,13,1>
     PRESERVE<0,6,7,8,9,10,11,12,14>
     NOPRESERVE<2,3,4,5>
    >
DEFINE PSECTOWN
    <
     .PSECT MYOWN
    >
DEFINE PSECTCODE
    <
     .PSECT MYCODE
    >
```

For additional information, refer to "BLISS-36 Linkage Declarations" in Chapter 13 of the BLISS Language Guide.

INDEX

BLISS-36
User's Guide
AA-H712D-TK

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03061

Do Not Tear - Fold Here

Cut Along Dotted Line