# VAX-11 BLISS-32
## User's Guide

Order No. AA-H322C-TE

**February 1982**

This document describes the VAX-11 BLISS-32 compiler and its use, and gives basic information about linking, executing, and debugging BLISS-32 programs. It also describes BLISS-32 machine-specific functions, BLISS tools, and other topics relevant to BLISS-32 programming.

**SUPERSESSION/UPDATE INFORMATION:** This document supersedes Version 2 of the BLISS-32 User's Guide, (Order No. AA-H322B-TE), dated January 1980.

**OPERATING SYSTEM AND VERSION:** VAX/VMS V2.5

**SOFTWARE VERSION:** BLISS-32 V3.0

digital equipment corporation · maynard, massachusetts

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

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

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



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 | RSX |
| DEC/CMS | EduSystem | UNIBUS |
| DECnet | IAS | VAX |
| DECsystem-10 | MASSBUS | VMS |
| DECSYSTEM-20 | PDP | VT |
| DECUS | PDT | digital |
| DECwriter | RSTS | |

ZK2203

CONTENTS

CONTENTS

CONTENTS



FIGURES



TABLES

# PREFACE

## MANUAL OBJECTIVES

This manual is a user's guide for the VAX-11 BLISS-32 compiler running under the VAX/VMS operating system. It is intended as a complement to the BLISS Language Guide. It provides three kinds of information: basic operating instructions, supplementary programming information, and reference material.

## INTENDED AUDIENCE

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

## STRUCTURE OF THIS DOCUMENT

Chapters 1 through 3 describe basic operating instructions.

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

Chapters 4 through 8 supply supplementary programming information.

- Chapter 4 defines the VAX-11 machine-specific functions.
- Chapter 5 discusses programming considerations, usch as the use of LIBRARY and REQUIRE facilities.
- Chapter 6 gives guidelines for writing transportable BLISS programs. Chapter 7 discusses the compiler
- Chapter 7 describes compiler architecture and the effects of the command qualifiers related to optimization.
- Chapter 8 describes some tools related to BLISS programming.

The appendixes contain reference material.

- Appendix A summarizes the command syntax, including the command qualifiers, their defaults, and abbreviations.
- Appendix B provides formatting rules.
- Appendix C provides a module template.
- Appendix D lists current implementation limits.
- Appendix E describes the error messages produced by the compiler.
- Appendix F is a sample output listing.

## ASSOCIATED DOCUMENTS

The VAX-11 Information Directory lists and describes all the documents that you may need to refer to in the course of building and executing a BLISS program.

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

| | |
|---|---|
| • BLISS language usage: | BLISS Language Guide |
| • BLISS syntax summary: | BLISS Pocket Guide |
| • Program linking and execution: | VAX-11 Linker Reference Manual |
| | VAX/VMS Command Language User's Guide |
| • Symbolic debugging: | VAX-11 Symbolic Debugger Reference Manual |
| • System services: | VAX/VMS System Services Reference Manual |
| | VAX-11 Record Management Services Reference Manual |
| | VAX-11 Record Management Services User's Guide |

## CONVENTIONS USED IN THIS DOCUMENT

Syntax notation and definitions used in BLISS-32 are explained thoroughly in Chapter 2 of the BLISS Language Guide. The following is a summary of syntax 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 repeated zero or more times with the items separated by spaces. |
| item,... | The item directly preceding the ",..." can be repeated zero or more times with the items separated by commas. |
| item+... | The item directly preceding the "+..." can be repeated zero or more times with the items separated by plus signs. |

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

## SUMMARY OF TECHNICAL CHANGES

This manual describes Version 3.0 of the BLISS-32 compiler. This section summarizes the technical changes in the use of the BLISS-32 since Version 2.0.

/ERROR_LIMIT has been added as a general-qualifier for the BLISS-32 command line.

The following machine-specific functions have been added.

**Arithmetic:**

| | | | | | |
|---|---|---|---|---|---|
| ADDD | ADDH | DIVF | MULD | MULH | SUBG |
| ADDF | ADDM | DIVG | MULF | SUBD | SUBH |
| ADDG | DIVD | DIVH | MULG | SUBF | SUBM |

**Arithmetic Conversion:**

| | | | |
|---|---|---|---|
| CVTDI | CVTGF | CVTHG | CVTRGL |
| CVTFG | CVTGH | CVTHL | CVTRGH |
| CVTFH | CVTGL | CVTLG | |
| CVTFI | CVTHF | CVTLH | |

**Arithmetic Comparison:**

| | | |
|---|---|---|
| CMPM | CMPG | CMPH |

**String Functions:**

| | | | |
|---|---|---|---|
| MOVC3 | CMPC3 | MOVTC | LOCC |
| MOVC5 | CMPC5 | MATCHC | SKPC |

**Additional BUILTIN Functions:**

| | |
|---|---|
| ASHP | XFC |

The following machine-specific functions will accept output parameters.

| | | | | | |
|---|---|---|---|---|---|
| CALLG | CRC | CVTPT | LOCC | MOVP | SKPC |
| CMPC3 | CVTLP | CVTSP | MATCHC | MOVTC | SPANC |
| CMPC5 | CVTPL | CVTTP | MOVC3 | MOVTUC | |
| CMPP | CVTPS | EDITPC | MOVC5 | SCANC | |

The following VAX/VMS DEBUG V3 commands are supported.

| | | |
|---|---|---|
| CANCEL SOURCE | SET SEARCH | SHOW SEARCH |
| SEARCH | SET SOURCE | SHOW SOURCE |
| SET MARGIN | SET STEP LINE | STEP LINE |
| SET MAX_SOURCE_FILES | SHOW MARGIN | TYPE |

## SUMMARY OF TECHNICAL CHANGES

The /DEBUG command line qualifier now generates source-line number information in the DEBUG symbol table when the compiler executes under VAX/VMS Version 3.0.

The compilation summary is now generated as part of the statistics.

The BLISS-32 compiler accepts quoted strings up to 1000 characters in length.

Appendix E includes changes in diagnostic messages. It has also been expanded to describe corrective actions.

# CHAPTER 1

# OPERATING PROCEDURES

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

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

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

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

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

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

## 1.1 COMPILING A BLISS MODULE

To compile a BLISS module, the programmer issues the BLISS system command and a BLISS command line. The command line consists of one or more source file names optionally preceded by command-line qualifiers. (Refer to "Command Line Syntax" below.) Some BLISS command line examples follow.

- To compile a module, give the following command:

```
$ BLISS MYPROG
```

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

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

```
$ BLISS/LIST MYPROG
```

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

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

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

  The BLISS compiler produces the object file, GAMMA.OBJ.

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

```
$ BLISS/LIBRARY ALPHA
```

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

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

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

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

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

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

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

### 1.1.1 Command-Line Syntax

| | |
|---|---|
| compilation-request | $ BLISS bliss-command-line |
| bliss-command-line | { qualifier,... } space input-spec,... |
| input-spec | file-spec{+...} {qualifier,...} |
| space | { blank \| tab } ... |
| qualifier | { output qualifier<br>general qualifier<br>terminal qualifier<br>optimization qualifier<br>source-list qualifier<br>machine-code-list qualifier } |

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

### 1.1.2 Command-Line Semantics

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

Unless a qualifier to change the compiler's behavior is given, the output from a compilation initiated from your terminal is the object file and the terminal listing, and the output from a compilation given in a batch file is an object file and listing file.

The compiler uses the contents of a file or of a series of files joined (concatenated) by plus signs (+) as input to a BLISS compilation. The compiler begins processing with the first file given in an input-spec and continues until an end-of-file is reached. It continues to read input until all files specified in the input-spec have been read. Command-line 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+THETA+ZETA,OMEGA
```

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

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

If no command-line 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 three contexts:

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

The file-spec is a standard VAX/VMS file specification, as described in the VAX/VMS Command Language User's Guide. (See Appendix A.)

A file specification is interpreted as follows:

1. Logical name translation occurs.

2. If a file-type is not given, a default file type is used, as described in the next section.

3. If the file-spec applies to an output file and a filename is not given, then the name of the first or only input file in the input-spec is used.

The compiler uses this same interpretation when it processes the file specification given in a REQUIRE or LIBRARY declaration. (Refer to Chapter 16 of the BLISS Language Manual The compiler has two ordered lists of default file types to be tried for an input-spec that does not include a file type. The default type the compiler applies depends on the output to be produced by the compilation, as indicated in the following list:

| **Input-spec used to Produce** | **Default Type List** |
|---|---|
| an object module | .B32, .BLI |
| a library file | .R32, .REQ, .B32, .BLI |

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

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

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

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

and you use the following command line to compile it:

```
$ BLISS ALPHA
```

Since the bliss-command-line contains no qualifier requesting that a library file be produced, the output of the compilation is an object module. Therefore, the compiler chooses the list of default types associated with object module output and searches first for ALPHA.B32, then, not finding that file, 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.R32, then CBLISS.REQ, then CBLISS.B32, then CBLISS.BLI. When the compiler processes the LIBRARY declaration, it uses the default type list associated with library declarations and searches for TBLISS.L32.

## 1.3 COMMAND-LINE QUALIFIERS

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

- output qualifier - defines the types of output to be produced
- general qualifier - sets a %VARIANT value and specifies code and debug information
- terminal qualifier - controls output produced on a terminal
- optimize qualifier - supplies code optimization strategies and directions
- source-list qualifier - provides output listing information concerning the form of the source part
- machine-code-list qualifier - provides output listing information concerning the form of the object part

### 1.3.1 Output Qualifiers

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

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

  ```
  $ BLISS/NOOBJECT ALPHA
  ```

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

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

  ```
  $ BLISS/LIST ALPHA
  ```

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

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

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

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

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

```
$ BLISS/LIBRARY ALPHA
```

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

**1.3.1.1 Syntax** - Output qualifier syntax is defined as follows:

| | |
|---|---|
| output qualifier | { /OBJECT {=file-spec} \| /NOOBJECT<br>/LIST {=file-spec} \| /NOLIST<br>/LIBRARY {=file-spec} \| /NOLIBRARY } |

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

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

```
/OBJECT   /NOLIST   /NOLIBRARY
```

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

```
/OBJECT   /LIST     /NOLIBRARY
```

If a file-spec is not given, the file name of the first file 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 default file-types are applied, depending on the file-designator:

| File-Designator | Default Type |
|---|---|
| /OBJECT | OBJ |
| /LIST | LIS |
| /LIBRARY | L32 |

**1.3.1.3 Semantics** - The output qualifiers 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.OBJ'. |
| /NOOBJECT | Do not produce an object file. |
| /LIST=file-spec | Produce a list file in the file specified by file-spec. |

| | |
|---|---|
| /LIST | Produce a list file in the file specified by 'input-file-name.LIS'. |
| /NOLIST | Do not produce a list file. |
| /LIBRARY=file-spec | Produce a library file in the file specified by file-spec. |
| /LIBRARY | Produce a library file in the file specified by 'input-file-name.L32'. |
| /NOLIBRARY | Do not produce a library file. |

### 1.3.2 General Qualifiers

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

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

  ```
  $ BLISS/NOTRACEBACK ALPHA
  ```

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

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

  ```
  $ BLISS/DEBUG ALPHA
  ```

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

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

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

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

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

**1.3.2.1 Syntax** - General qualifier syntax is defined as follows:

| | |
|---|---|
| general qualifier | ( /TRACEBACK \| /NOTRACEBACK )<br>( /DEBUG \| /NODEBUG )<br>( /CODE \| /NOCODE )<br>( /VARIANT { =value } )<br>( /ERROR_LIMIT { =value } ) |

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

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

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

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

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

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

| | |
|---|---|
| /TRACEBACK | Generate information in the object module that can be used by the VAX-11 Debugger to locate module, routine, and program section (PSECT) names. |
| /NOTRACEBACK | Produce the minimum size object module. Do not include any information for debugging or tracing. |
| /DEBUG | Generate information in the object module that can be used by the VAX-11 Debugger to reference names declared within the BLISS module. |
| /NODEBUG | Do not generate any additional debug information. If this switch is applied either explicitly or by default, the VAX-11 Debugger can only locate module, routine, and PSECT names. |
| /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: $-(2^{**}31) \leq n \leq (2^{**}31)-1$ |
| /ERROR_LIMIT | Set limit to 1 |
| /ERROR_LIMIT=n | Terminates compilation after n error level diagnostics are encountered. |

**1.3.2.4 Discussion** - An object module can be produced with the following degrees of information for the linker, debugger, and the operating system.

- No information (/NOTRACEBACK)
- Basic information about modules, routines, and PSECTs (/TRACEBACK and /NODEBUG)

- Information about modules, routines, PSECTs, and data-segment names (/TRACEBACK and /DEBUG)

The default object module contains the basic information. For example, if a program fails because of an access violation, VMS can display the state of the program when the violation occurred through its traceback facility. Further, the VAX-11 Debugger can be used to refer to modules, routines, and PSECTs symbolically and to call routines.

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

### 1.3.3 Terminal Qualifier

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

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

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

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

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

**1.3.3.1 Syntax** - Terminal qualifier syntax is defined as follows:

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

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

```
ERRORS  NOSTATISTICS
```

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

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

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

### 1.3.4 Optimize Qualifier

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

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

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

  Note that use of QUICK turns off flow analysis. (Refer to Section 7.1.2.)

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

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

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

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

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

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

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

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

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

**1.3.4.1 Syntax** - Optimize qualifier syntax is defined as follows:

| | |
|---|---|
| optimize qualifier | /OPTIMIZE= { ( optimize-value ,... ) / optimize-value } |
| optimize-value | { QUICK \| NOQUICK / SPEED \| SPACE / LEVEL : optimize-level / SAFE \| NOSAFE } |
| optimize-level | { 0 \| 1 \| 2 \| 3 } |

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

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

NOQUICK SPACE LEVEL:2 SAFE

The compiler is directed:

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

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

| Optimize-Value | Meaning |
|---|---|
| QUICK | Omit some standard optimizations (for example, turn off flow analysis) in order to increase compilation speed. |
| NOQUICK | Perform standard optimizations. |
| SPEED | Increase the potential execution speed of the program being compiled (if possible) by using more space where necessary. For more information on the effect of this value, see Section 7.1.4. (Note: SPEED is equivalent to the module-switch ZIP.) |
| SPACE | Keep program size to a minimum at the possible expense of operating speed. For more information on the effect of this value, see Section 7.1.4. (Note: SPACE is equivalent to the module-switch NOZIP.) |

LEVEL

Optimize the program being compiled according to the optimize-level given, as follows:

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

LEVEL:3 optimizes speed at the expense of space in the same way as SPEED. For more information on the effect of this value, see Section 7.2.

SAFE

Assume that all named data-segments are referenced by name only and not manipulated indirectly in any way, and use optimization techniques that exploit this fact. For more information on the effect of this value, see Section 7.1.2.1.

NOSAFE

Assume that sometimes a named data-segment is referenced by means of a computed expression and, therefore, some optimization techniques cannot be used.

### 1.3.5 Source-List Qualifier

The source-list qualifier is used to supply information about the form of the source part of the output listing. Some examples of the use of the source-list qualifier are as follows:

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

```
$ BLISS/LIST/SOURCE_LIST=PAGE SIZE:44 ALPHA
```

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

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

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

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

**1.3.5.1 Syntax** - Source-list qualifier syntax is defined as follows:

| | |
|---|---|
| source-list qualifier | /SOURCE_LIST= { ( source-value ,... ) \| source-value } |
| source-value | { HEADER \| NOHEADER<br>PAGE_SIZE : number-of-lines<br>LIBRARY \| NOLIBRARY<br>REQUIRE \| NOREQUIRE<br>EXPAND_MACROS \| NOEXPAND_MACROS<br>TRACE_MACROS \| NOTRACE_MACROS<br>SOURCE \| NOSOURCE } |
| number-of-lines | { 20 \| 21 \| 22 \| ... } |

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

```
HEADER  PAGE_SIZE:58  NOLIBRARY  NOREQUIRE
NOEXPAND MACROS  NOTRACE_MACROS  SOURCE
```

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

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

| Source-Value | Meaning |
|---|---|
| HEADER | Page the listing produced on the list file and include a heading on each page. |
| NOHEADER | Do not page the listing, do not include headings, and do not produce statistics in the compilation summary. Use this value if the assembleable listing is to be assembled. |
| PAGE_SIZE:lines | Use the number of lines specified for each page of the list file. The number of lines must be greater than 19. |
| LIBRARY | Produce a trace in the listing file identifying the library after a LIBRARY declaration and the first use of each name whose definition is obtained from a library file. For an example of a library trace, see Section 2.2.4.2. |
| NOLIBRARY | Do not produce a trace identifying any libraries and their contributions. |
| REQUIRE | Include source text obtained from require files in listing. |

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

### 1.3.6 Machine-Code-List Qualifier

The machine-code-list qualifier is used to give the compiler instructions about the form of the object part of the output listing. Some examples of the use of the machine-code-list qualifier are as follows:

- To obtain an output listing that can be subsequently edited and then reassembled by the VAX-11 MACRO assembler, use the ASSEMBLER code-value, as follows:

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

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

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

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

**1.3.6.1 Syntax** - Machine-code-list qualifier syntax is defined as follows:

| machine-code-list qualifier | /MACHINE_CODE_LIST= { ( code-value ,... ) \| code-value } |
|---|---|
| code-value | { OBJECT \| NOOBJECT<br>ASSEMBLER \| NOASSEMBLER<br>SYMBOLIC \| NOSYMBOLIC<br>BINARY \| NOBINARY<br>COMMENTARY \| NOCOMMENTARY<br>UNIQUE_NAMES \| NOUNIQUE_NAMES } |

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

OBJECT NOASSEMBLER SYMBOLIC BINARY COMMENTARY NOUNIQUE_NAMES

The compiler produces a listing that resembles the output listing of the VAX-11 MACRO assembler.

**1.3.6.3 Semantics** - The /MACHINE_CODE_LIST qualifier indicates that one or more code-values follow. The code-values have the following meanings:

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

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

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

**Table 1-1: Correspondence Between Qualifier and Switch Names**

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

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

### 1.3.8 Qualifiers and Default Settings

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

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

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

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

### 1.3.9 Positive and Negative Forms of Qualifiers

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

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

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

### 1.3.10 Abbreviations of Qualifier and Value Names

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

# CHAPTER 2

# COMPILER OUTPUT

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

The input to a BLISS compilation is a BLISS module. As an example, consider the following module: it contains two OWN declarations and three ROUTINE declarations. The routine IFACT computes the factorial of its argument by an iterative method. The routine RFACT computes the factorial of its argument by a recursive method. The routine MAINPROG provides some test calls on IFACT and RFACT. Factorial routines are discussed in Chapter 12, "Routines," in 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.

## 2.1 TERMINAL OUTPUT

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

Error messages show the source program line associated with the error followed by a description of the error. The statistics show the name of each routine declaration in the module and the number of bytes associated with that declaration. The compilation statistics give the number of warning and error messages, the number of code bytes and data bytes used by the program, the run time and elapsed time required for the compilation, and the number of pages of VAX-11 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.B32. To obtain both kinds of information, compile the module by using the following bliss-command-line:

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

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

```
;       0002    begin
% WARN#048       1  L1:0002
; Syntax error in module head
;       0014            result = .reult*.I;
% WARN#000       ................1  L1:0014
; Undeclared name:  REULT
IFACT 22
RFACT 26
MAINPROG 28
; Warnings:      2
; Errors:        0
; Size:          76 code + 8 data bytes
; Run time:      00:01.5
; Elapsed time:  00:03.5
; Memory used:   10 pages
; Compilation Complete
%BLS32-W-CMPWARN, Compilation with warnings
```

Terminal output for the compilation of MYPROG includes two warnings, which are described later. Statistics following the warnings show the number of bytes required for each routine. The module TESTFACT contains three routine declarations, namely, IFACT, RFACT, and MAINPROG, and they use 22, 26, and 28 bytes, respectively. The compilation summary shows that the compilation of TESTFACT required 1.5 seconds of processor time and that 3.5 seconds elapsed. The compilation required nine pages of memory, excluding memory required for the compiler itself.

## 2.2 OUTPUT LISTING

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

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

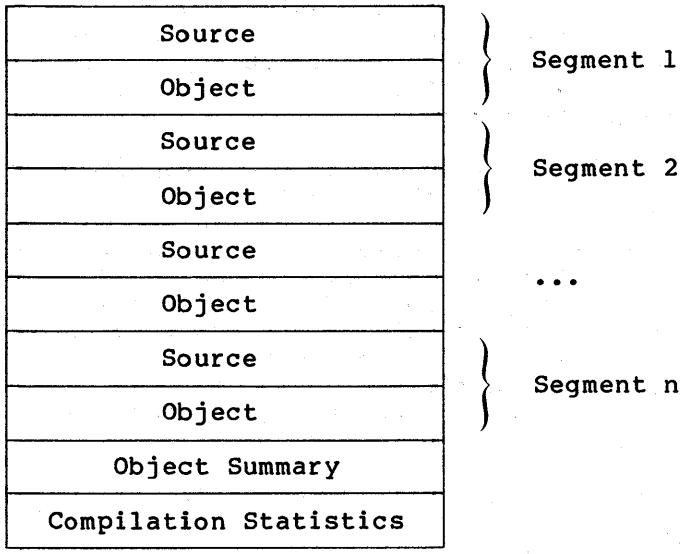


**Figure 2-1: Compiler Output Listing Sequence**

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

The complete output listing for the module TESTFACT occupies several pages. (Refer to "Default Object Listing" later in this chapter.) 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.

### 2.2.1 Listing Header

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

| Print Position | 1 | 15 | 17 | 63 | 65 | 132 |
|---|---|---|---|---|---|---|
| | name | | title | | processor identification | |
| | ident | | subtitle | | source identification | |

**Figure 2-2: 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, t include the information if it appears in the object module. If the module name exceeds 15 characters, the title field begins eight 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 the user 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 the user updates the title or subtitle information in the first line of the source page, the listing for that page will include the updated information.

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

The source identification field contains the data 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.

### 2.2.2 Source Listing

The source part of the output listing reproduces the input to the BLISS compilation with annotation supplied by the compiler. The compiler annotation includes a 14-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 14-character preface string consists of a semicolon (;), followed by either the editor line sequence number for a sequenced file or five blanks for an unsequenced file, two 1-column codes, a 4-digit line number generated by the compiler, and two blanks.

Thus, the preface string has the following general form:

```
;xxxxxyznnnnbb
```

Table 2-1 describes preface string components.

**Table 2-1: 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 VAX-11 MACRO assembler. |
| xxxxx | 2-6 | The line number, if the file contains line sequence numbers; otherwise, five blank columns. |
| y | 7 | A code that indicates the lexical processing level of the compiler. The codes that can appear in this column are described below:<br><br>**Code** — **Meaning**<br><br>C — Embedded comment, that is, text within %(...)%.<br>D — Default lexeme stream for a keyword macro formal.<br>L — Parameter list of a lexical function.<br>M — Body of a macro definition.<br>P — Parameter list of a macro call.<br>U — Source text which is discarded by an unsatisfied lexical condition.<br><br>If more than one such code applies (for example, an embedded comment nested within a macro body), the "innermost" code is printed. |
| z | 8 | If the line comes from a file specified in a REQUIRE declaration, the code "R"; otherwise, blank. |
| nnnn | 9-12 | The BLISS line sequence number, beginning with 0001 and 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. |
| bb | 13-14 | Blank |

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

```
RESULT = 1;
```

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

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

The line number 0014 is the line assigned by the BLISS compiler. If the input line had an editor sequence number 2300, the output listing for that line would be::

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

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

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

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

```
;     M 0004  RESULT = 1;
```

The y item in the preface string (column 7) 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 finding the point at which the M code first appears in the y field before the runaway.

An example of the source listing for the first segment of the module TESTFACT appears below:

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

Following three heading lines, the source of the module TESTFACT is reproduced. The preface string begins with a semicolon (;). If the input file that contains the module TESTFACT does not have sequence numbers, columns 2 through 6 of the source listing are blank. Columns 7 and 8 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 9.

Two error messages are reported as part of the source listing. Section 2.4 contains a discussion of error messages in general and of the meaning of these errors in particular.

### 2.2.3 Object Listing

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

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

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

```
                 ( SYMBOLIC   )                 ( BINARY   )  ( UNIQUE_NAMES   )
ASSEMBLER        (            )  COMMENTARY     (          )  (                 )
                 ( NOSYMBOLIC )                 ( NOBINARY )  ( NOUNIQUE_NAMES )

                                                ( BINARY   )
NOASSEMBLER  SYMBOLIC   COMMENTARY              (          )  NOUNIQUE_NAMES
                                                ( NOBINARY )
```

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

The question of whether or not to have the binary appear on the listing is a question of personal preference. However, it may at times be useful for debugging purposes. If the binary field is omitted, the listing is likely to be more compressed, since additional operands can then be placed on the same line.

The compiler produces the following information for each field.

ASSEMBLER field — Instructions in assembler form, for example:

```
MOVL    #1,R0
```

SYMBOLIC field — Instructions in assembler form, but using symbolic source names, for example:

```
MOVL    #1,RESULT
```

BINARY field

Hexadecimal equivalent of instructions and data to enable easier debugging. The hexadecimal instructions appear in the same format as that produced by the VAX-11 MACRO assembler as far as possible. The rightmost numeric field in the binary listing is the location counter, relative to the starting routine's address. This simplifies user interaction with VAX-11 DEBUG when setting breakpoints or examining instructions. (Refer to Section 3.3, "Debugging.")

The following codes are included in the hexadecimal information in the binary field to provide information about relocation of quantities:

| Code | Meaning |
|---|---|
| Blank | Absolute quantity (no linker action) |
| V | Forward relocatable. |
| * | Complex |
| ' | Relocated (relative to a program section other than the code program section) |
| G | either general addressing (Position Independent) or globally relocatable |

COMMENTARY field

A cross-reference to the source program line generating the code. If a program line generates more than one instruction line, commentary fields in the lines following the first generated instruction remain blank.

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

**2.2.3.1 Default Object Listing** - The listing in Figure 2-3 was produced by the following command line:

```
$ BLISS/LIST MYPROG
```

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

```
TESTFACT                                        31-Oct-1979 10:34:06    VAX-11 Bliss-32 T2-619          Page   1
                                                31-Oct-1979 10:28:35    DBB1:[DIRECTORY]MYPROG.BLI;5 (1)

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


                                                                .TITLE  TESTFACT

                                                                .PSECT  $OWN$,NOEXE,2

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

                                                                .EXTRN  REULT

                                                                .PSECT  $CODE$,NOWRT,2

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

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


;01700  0017     routine rfact (n) =
;01900  0018         if .n gtr 1 then .n*rfact (.n - 1) else 1;



                                                  0000 00000 RFACT:     .WORD   Save nothing                    ; 0017
                                 01         04   AC  D1 00002        CMPL    N, #1                           ; 0018
                                                  0E  15 00006        BLEQ    1$                              ;
                        7E       04        AC       01  C3 00008        SUBL3   #1, N, -(SP)                    ;
                                 EF        AF       01  FB 0000D        CALLS   #1, RFACT                       ;
                                 50        04   AC  C4 00011        MULL2   N, R0                           ;
                                                     04 00015        RET                                     ;
```

Figure 2-3:  Default Object Listing Example

```
TESTFACT                                                31-Oct-1979 10:34:06    VAX-11 Bliss-32 T2-619              Page   2
                                                        31-Oct-1979 10:28:35    DBB1:[DIRECTORY]MYPROG.BLI;5 (1)

                                 50            01  D0 00016 1$:     MOVL    #1, R0                                  ;
                                                   04 00019         RET                                             ; 0017

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


;02000  0019
;02100  0020     routine mainprog : novalue =
;02200  0021         begin
;02300  0022         a = ifact (5);
;02400  0023         b = rfact (5);
;02500  0024         end;



                                                  0000 00000 MAINPROG:
                                                             .WORD   Save nothing                                    ; 0020
                                                   05  DD 00002         PUSHL   #5                                  ; 0022
                                     C8   AF       01  FB 00004         CALLS   #1, IFACT                           ;
                                   0000'  CF       50  D0 00008         MOVL    R0, A                               ;
                                                   05  DD 0000D         PUSHL   #5                                  ; 0023
                                     D3   AF       01  FB 0000F         CALLS   #1, RFACT                           ;
                                   0000'  CF       50  D0 00013         MOVL    R0, B                               ;
                                                       04 00018         RET                                         ; 0020

; Routine Size:  25 bytes,    Routine Base:  $CODE$ + 0030


;02550  0025
;02600  0026     end
;02800  0027     eludom



;                                       PSECT SUMMARY
;
;       Name                    Bytes                           Attributes
;
;  $OWN$                           8       ,  WRT,  RD ,NOEXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)
;  $CODE$                         73       ,NOWRT,  RD ,  EXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)



; Warnings:     2
; Errors:       0


;                                       COMMAND QUALIFIERS

;       BLISS /LIST MYPROG
```

Figure 2-3 (Cont.): Default Object Listing Example

```
; Size:         73 code + 8 data bytes
; Run Time:     00:01.5
; Elapsed Time: 00:03.0
; Memory Used:  9 pages
; Compilation Complete
```

**Figure 2-3 (Cont.): Default Object Listing Example**

**2.2.3.2 Assembler Input Listing** - The listing in Figure 2-4 was produced by compiling module TESTFACT with the following code-options:

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

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

## 2.2.4 Source Part Options

The following sections contain more output listings to illustrate different options for the source part of the list file. To illustrate these different forms, the sample program TESTFACT has to be made more 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 the production of a new module TEST that uses the factorial routine to take combinations, according to the following formula for obtaining the number of combinations of m things taken n at a time:

$$\left\{ \begin{matrix} m \\ n \end{matrix} \right\} = \frac{n!}{(m-n)!\ m!}$$

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

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

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

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

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

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

```
$ BLISS/LIBRARY COMBN
```

Finally, include some test combinations.

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

```
TESTFACT                                        31-Oct-1979 10:45:47    VAX-11 Bliss-32 T2-619              Page   1
                                                31-Oct-1979 10:28:35    DBB1:[DIRECTORY]MYPROG.BLI;5 (1)

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


        .TITLE  TESTFACT

        .PSECT  $OWN$,NOEXE,2

A:      .bKB    4
B:      .bKB    4

        .EXTRN  REULT

        .PSECT  $CODE$,NOWRT,2

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

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


;01700  0017    routine rfact (n) =
;01900  0018        if .n gtr 1 then .n*rfact (.n - 1) else 1;



RFACT:  .WORD   ^M<>                    ;Save nothing                           ; 0017
        CMPL    4(AP), #1               ;N, #1                                  ; 0018
        BLEQ    1$                      ;1$                                     ;
        SUBL3   #1, 4(AP), -(SP)        ;#1, N, -(SP)                           ;
        CALLS   #1, B^RFACT             ;#1, RFACT                              ;
        MULL2   4(AP), R0               ;N, R0                                  ;
        RET                             ;                                       ;
```

**Figure 2-4: Assembler Input Listing Example**

```
TESTFACT                                                31-Oct-1979 10:45:47    VAX-11 Bliss-32 T2-619              Page   2
                                                        31-Oct-1979 10:28:35    DBB1:[DIRECTORY]MYPROG.BLI;5 (1)

1$:     MOVL    #1, R0                          ;#1, R0                                                         ;
        RET                                     ;                                                               ; 0017

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


;02000  0019
;02100  0020     routine mainprog : novalue =
;02200  0021         begin
;02300  0022         a = ifact (5);
;02400  0023         b = rfact (5);
;02500  0024         end;



MAINPROG:
        .WORD   ^M<>                            ;Save nothing                                                   ; 0020
        PUSHL   #5                              ;#5                                                             ; 0022
        CALLS   #1, B^IFACT                     ;#1, IFACT                                                      ;
        MOVL    R0, W^A                         ;R0, A                                                          ;
        PUSHL   #5                              ;#5                                                             ; 0023
        CALLS   #1, B^RFACT                     ;#1, RFACT                                                      ;
        MOVL    R0, W^B                         ;R0, B                                                          ;
        RET                                     ;                                                               ; 0020

; Routine Size:  25 bytes,    Routine Base:  $CODE$ + 0030


;02550  0025
;02600  0026     end
;02800  0027     eludom



;                                       PSECT SUMMARY
;
;       Name                            Bytes                           Attributes
;
;  $OWN$                                   8       ,   WRT,  RD ,NOEXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)
;  $CODE$                                 73       ,NOWRT,  RD ,  EXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)



; Warnings:      2
; Errors:        0


;                                       COMMAND QUALIFIERS

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

**Figure 2-4 (Cont.): Assembler Input Listing Example**

```
TESTFACT                                31-Oct-1979 10:45:47    VAX-11 Bliss-32 T2-619              Page   3

; Size:         73 code + 8 data bytes
; Run Time:     00:01.6
; Elapsed Time: 00:04.0
; Memory Used:  10 pages
; Compilation Complete

        .END    MAINPROG
```

**Figure 2-4 (Cont.): Assembler Input Listing Example**

**2.2.4.1 Default Source Listing** - The following command line generated the output listing in Figure 2-5 for the module TEST:

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

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

```
                .
                . (header)
                .
;           0001  module test (main = mainprog) =
;           0002  begin
;           0003
;           0004  own
;           0005      a,
;           0006      b;
;           0007  external routine
;           0008      error;
;           0009  library 'combn' ;
;           0010  require 'rfact' ;
;           0017
;           0018  routine mainprog =
;           0019      begin
;           0020      a = combinations (3, 2);
;           0021      b = combinations (6, 4);
;           0022      end;
;           0023
;           0024  end
;           0025  eludom



;                                   LIBRARY STATISTICS
;
;                                   -------- Symbols --------     Blocks
;         File                      Total    Loaded   Percent       Read
;
;  DBB1:[DIRECTORY]COMBN.L32;3        2         2       100           4




;                                             COMMAND QUALIFIERS

;         BLISS /LIST/NOCODE TEST

; Run Time:       00:00.7
; Elapsed Time:   00:03.6
; Memory Used:    8 pages
; Compilation Complete
```

**Figure 2-5: Default Source Listing Example**

**2.2.4.2 Listing with LIBRARY and REQUIRE Information** - The command line:

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

generated the output listing in Figure 2-6, which contains information from 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 0020. The contents of the REQUIRE file are given in lines 0011 through 0016.

**2.2.4.3 Listing with Macro Expansions** - The command line:

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

generated the output listing in Figure 2-7 to illustrate macro expansions, which follow lines 0020 and 0021. Observe that expansions are listed in the order in which they occur. The innermost expansion is printed first, followed by the outer expansion, which includes the expanded form of the inner macro. Therefore, the last line of the macro expansion is the fully expanded form.

**2.2.4.4 Listing with Macro Tracing** - The command line:

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

produced the output listing in Figure 2-8, which contains macro tracing and macro expansion information. The macro trace gives information about parameter binding in addition to the expansion information.

## 2.3 COMPILATION SUMMARY

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

- The routine size and psect-relative starting address (following each routine)
- A program section summary (at the end of the module)
- Library usage statistics indicating the libraries used and the number of names loaded from each library (omitted if no libraries are used)
- 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, memory used, and a statement that the compilation is complete.

```
TEST                                                    31-Oct-1979 15:32:08    VAX-11 Bliss-32 T2-619              Page   1
                                                        31-Oct-1979 12:37:24    DBB1:[DIRECTORY]TEST.BLI;5 (1)

;          0001  module test (main = mainprog) =
;          0002  begin
;          0003
;          0004  own
;          0005      a,
;          0006      b;
;          0007  external routine
;          0008      error;
;          0009  library 'combn' ;
;; Library file DBB1:[DIRECTORY]COMBN.L32;3 produced by VAX-11 Bliss-32 T2-619 on 31-Oct-1979 15:13:25
;          0010  require 'rfact' ;

;         R0011  routine fact (n) =
;         R0012      if .n gtr 1
;         R0013      then
;         R0014          .n*fact (.n - 1)
;         R0015      else
;         R0016          1;

;          0017
;          0018  routine mainprog =
;          0019      begin
;          0020      a = combinations (3, 2);
;; Loaded symbol COMBINATIONS from library DBB1:[DIRECTORY]COMBN.L32;3
;; Loaded symbol COMB from library DBB1:[DIRECTORY]COMBN.L32;3
;          0021      b = combinations (6, 4);
;          0022      end;


;          0023
;          0024  end
;          0025  eludom



;                                LIBRARY STATISTICS
;
;                                          -------- Symbols --------     Blocks
;       File                               Total    Loaded   Percent       Read
;
;  DBB1:[DIRECTORY]COMBN.L32;3                 2         2       100          4



;                                       COMMAND QUALIFIERS

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

; Run Time:      00:00.6
; Elapsed Time:  00:02.5
; Memory Used:   8 pages
; Compilation Complete
```

**Figure 2-6: Output Listing Example Showing Library and Require File Information**

```
TEST                                                    31-Oct-1979 15:41:30    VAX-11 Bliss-32 T2-619              Page   1
                                                        31-Oct-1979 12:37:24    DBB1:[DIRECTORY]TEST.bI;5 (1)

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


;          0017
;          0018  routine mainprog =
;          0019      begin
;          0020      a = combinations (3, 2);
;;        [COMB]= FACT ( 3 ) / ( FACT ( 3 - 2 ) * FACT ( 2 ) )
;;     [COMBINATIONS]= ( IF 3 LSS 2 THEN ERROR ( ) ELSE FACT ( 3 ) / ( FACT ( 3 - 2 ) * FACT ( 2 ) ) )
;          0021      b = combinations (6, 4);
;;        [COMB]= FACT ( 6 ) / ( FACT ( 6 - 4 ) * FACT ( 4 ) )
;;     [COMBINATIONS]= ( IF 6 LSS 4 THEN ERROR ( ) ELSE FACT ( 6 ) / ( FACT ( 6 - 4 ) * FACT ( 4 ) ) )
;          0022      end;


;          0023
;          0024  end
;          0025  eludom



;                                LIBRARY STATISTICS
;
;                                          -------- Symbols --------     Blocks
;       File                               Total    Loaded   Percent      Read
;
;  DBB1:[DIRECTORY]COMBN.L32;3                 2         2       100         4




;                                       COMMAND QUALIFIERS

;       BLISS /NOCODE/LIST/SOURCE_LIST:EXPAND_MACROS TEST

; Run Time:     00:00.6
; Elapsed Time: 00:03.1
; Memory Used:  8 pages
; Compilation Complete
```

Figure 2-7: Output Listing Example Showing Macro Expansion Information

```
TEST                                                    31-Oct-1979 15:59:16    VAX-11 Bliss-32 T2-619              Page   1
                                                        31-Oct-1979 12:37:24    DBB1:[DIRECTORY]TEST.BLI;5 (1)

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



;                              LIBRARY STATISTICS
;
;                                          -------- Symbols --------    Blocks
;      File                                Total    Loaded   Percent      Read
;
;  DBB1:[DIRECTORY]COMBN.L32;3                 2         2       100         4

; Run Time:     00:00.7
; Elapsed Time: 00:01.8
; Memory Used:  8 pages
; Compilation Complete
```

**Figure 2-8: Output Listing Example Showing Macro Expansion and Tracing Information**

## 2.4 ERROR MESSAGES

The BLISS compiler detects two types of errors, namely: 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 following warning message:

```
;       0014    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:

```
INCR I FROM 2 TO .NDO
RESULT = .REULT*.I;
```

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

```
;       0013    INCR I FROM 2 TO .NDO
; WARN #000     ................1   L1:0013
; Undeclared name: NDO
;       0014      RESULT = .REULT*.I;
; ERR #066 ............1  L1:0014  L2:0014  L3:0014
; Two consecutive operands with no intervening operator
```

Omitting the blank between the name N and the keyword DO caused one warning and one fatal error. Because in the absence of a separator the compiler sees NDO as a single name, it first identifies it as an undefined name. When it fails to find the keyword DO, the compiler cannot make syntactic sense out of the lines; therefore, it must report a fatal error message and suppress the production of any object output.

Observe that although the compiler continues to check the syntax of the remainder of the module, it misses the error REULT, because some text must be left unscanned to recover from the fatal error. The recovery process sometimes causes genuine errors to be missed and spurious errors to be reported. A module cannot be assumed to be fully checked by the compiler until all error messages are eliminated.

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

```
;       0013    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 2-9. This version of TESTFACT includes the coding error illustrated in the ccove examples. The error message at the end of the program identifies with line indicators the point at which tHe errorwas 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    MODULE TESTFACT (MAIN = MAINPROG)
;         0002    BEGIN
; WARN#048        1  L1:0002
; Syntax error in module head
;         0003
;         0004    OWN
;         0005        A,
;         0006        B;
;         0007
;         0008    ROUTINE IFACT (N) =
;         0009        BEGIN
;         0010        LOCAL
;         0011            RESULT;
;         0012        RESULT = 1;
;         0013        INCR I FROM 2 TO .NDO
; WARN#000        ......................1  L1:0013
; Undeclared name:  NDO
;         0014                RESULT = .REULT*.I;
; ERR #066        .........1  L1:0014  L2:0014  L3:0014
; Two consecutive operands with no intervening operator
;         0015        .RESULT
;         0016        END;
;         0017
;         0018    ROUTINE RFACT (N) =
;         0019        IF .N GTR 1
;         0020        THEN
;         0021             .N*RFACT (.N - 1)
;         0022        ELSE
;         0023             1;
;         0024
;         0025    ROUTINE MAINPROG =
;         0026        BEGIN
;         0027        A = IFACT (5);
;         0028        B = RFACT (5);
;         0029        END;
;         0030
;         0031    END
;         0032    ELUDOM
; ERR #012        1...2  L1:0032  L2:0013  L3:0032
; Missing DO following INCR or DECR


; Warnings:       2
; Errors:         2
; Size:           0 code + 8 data bytes
; Run Time:       00:02
; Elapsed Time:   00:03
; Memory Used:    3K
; Compilation Complete
```

**Figure 2-9: Error Messages in Source Listing Example**

# CHAPTER 3

# LINKING, EXECUTING, AND DEBUGGING

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

## 3.1 LINKING

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

Some examples of linking are:

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

  ```
  $ LINK ALPHA
  ```

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

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

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

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

The link operation is described in detail in the VAX-11 Linker Reference Manual.

### 3.1.1 LINK Command Syntax

Syntax of the LINK command is defined as follows:

| | |
|---|---|
| link-command | LINK { /DEBUG / nothing } space object-file ,... |
| object-file | file-spec |

### 3.1.2 LINK Command Semantics

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

If the /DEBUG qualifier is used in the link-command, the VAX-11 Debugger is linked with the specified object-files.

The linker has many qualifiers, but only the /DEBUG qualifier is discussed here. Other qualifiers are described in the VAX-11 Linker Reference Manual.

Appearance of the following error message during the link operation:

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

generally implies that the program is larger than 32KB. Specifying the module qualifier:

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

causes the compiler to generate long displacement values for instruction operands.

## 3.2 EXECUTING A LINKED PROGRAM

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

```
$ RUN ALPHA
```

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

## 3.3 DEBUGGING

The VAX-11 Symbolic Debugger can be used to debug or test BLISS programs. The following discussion of BLISS debugging assumes that you are familiar with the use of the debugger for VAX-11 MACRO programs, as described in the VAX-11 Symbolic Debugger Reference Manual. This section describes BLISS debugging facilities primarily in terms of differences from MACRO-level debugger usage.

The debugger recognizes BLISS as one of its "known" languages (as it does MACRO and FORTRAN, for example), and provides a number of features specifically designed for more convenient debugging of BLISS programs. These BLISS-specific features are:

- Debug expression syntax that is consistent with BLISS language syntax: radix control operators, arithmetic- and address-expression operators, field selectors, and so on.

- Support of fetch operator in arithmetic expressions.
- Expression evaluation according to BLISS rules for operator precedence and use of parentheses.
- BLISS-style references to elements and components of standard (predeclared) data structures: VECTOR, BLOCK, BLOCKVECTOR, and BITVECTOR; and support of REF data segments.
- Support of field-names in structure references.
- Support of BLISS field-references, as in: EXAMINE X<15,3,0>.

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

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

### 3.3.1 Initialized Modes and Types

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

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

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

```
SYMBOLIC, HEXADECIMAL
```

These mode settings can be changed with the SET MODE command, and can be restored to the initial settings with the CANCEL MODE or SET LANGUAGE BLISS commands. The initialized modes are the same as for MACRO.

The initial setting for default type is:

```
LONG INTEGER
```

(Refer to the SET TYPE command.)

The initial scope is PC scope.

### 3.3.2 Debug Commands and Expression Syntax

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

Experienced VAX-11 Symbolic Debugger users should note in particular that, in expressions, the BLISS dot symbol (.) is recognized instead of the "@" character as the debugger "contents" or indirect operator, and that the previous location symbol (^) is not recognized in BLISS mode.

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

Note that the significance of special characters as delimiters in debug commands is the same for both BLISS and MACRO debugging. Also in order to maintain a reasonable correspondence with the VAX-11 Symbolic Debugger Reference Manual, the informal categories "arithmetic expression" and "address expression" are used below (on a best-fit basis) rather than the more formal BLISS language-syntax categories.

### 3.3.3 Operators in Arithmetic Expressions

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

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

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

Table 3-1: Arithmetic Expression Operators

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

### 3.3.4 Special Characters in Address Expressions

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

Table 3-2: Address Representation Characters

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

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

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

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

will display the address of INBUF (that is, the contents of location PTR), and the command:

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

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

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

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

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

**3.3.4.2 Last Value Displayed Symbol (\)** - The backslash character (\) can be used to represent the value most recently displayed. The value remains unchanged until the debugger displays a new value.

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

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

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

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

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

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

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

Other examples, modified from those given in Chapter 4 of the VAX-11 Symbolic Debugger Reference Manual, follow. The command

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

shifts the current contents of the location MASK four bit positions to the left.

The command

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

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

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

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

Several examples of range-operator usage, modified from those given in Chapter 4 of the VAX-11 Symbolic Debugger Reference Manual, follow:

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

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

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

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

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

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

**3.3.4.5 Default Next-Location Value** - The default next-location value is the address implied by the "shorthand" form of the EXAMINE command, that is, the EXAMINE verb (and possibly a type specification) immediately followed by a carriage return. The next-location value is always the address of the byte following the last byte of the previous display item.

Continuing with the previous example involving INBUF, if subsequent to the command

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

and its resultant display, the "shorthand" command

```
DBG>EXAMINE/ASCII:1
```

is given, the display

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

would be produced, or the display

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

if type ASCII:2 was specified, or

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

if type ASCII:4 was specified.

### 3.3.5 Field References

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

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

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

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

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

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

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

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

In an EVALUATE command, the meaning of a field selector depends upon whether or not a contents operator is specified, that is, whether or not indirection is requested. If the contents operator is specified, a field selector has the same meaning (and lack of restriction on position) as in the BLISS fetch context. For example:

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

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

If indirection is **not** specified in an EVALUATE command, a field selector has the same meaning and restrictions as in the BLISS nonfetch, nonassignment content; that is, the expression

```
addr<p,s>
```

simply calculates the value addr+(p/8), where **p** must be a multiple of 8 and **s** is ignored. The effect of a field selector in this case is simply that of an address offset, with the position value (p) interpreted as p/8. For example, assuming (as above) that the symbol INBUF has the value 1024:

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

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

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

If a field selector appears in a "data" (righthand) operand of a DEPOSIT command, it has the same meaning and effect as described above for the EVALUATE command, depending upon whether or not a contents operator is specified.

### 3.3.6 Structure References

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

- For VECTOR-structured segments -

  segname[actual_1] or segname[field-name]

- For BITVECTOR-structured segments -

  segname[actual_1] or segname[field-name]

- For BLOCK-structured segments -

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

- For BLOCKVECTOR-structured segments -

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

where

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

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

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

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

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

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

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

1. Displays the content of the component specified by the first structure reference.

2. Skips any remaining bits (up to seven) if the component does not end on a byte boundary.

3. Displays any remaining data within the range in 2-byte units beginning with the next location, and ending with the last location occupied by any part of the component specified in the second structure reference (plus an additional byte if required to round up to a word-size display unit).

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

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

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

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

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

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

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

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

This example illustrates two facts:

- That the EVALUATE command evaluates a field name by reporting the access-actual values defined for that name. (Access-actual values, like field-selector parameters, are always displayed in decimal regardless of current radix mode.) The last-value-displayed (represented by \) following this type of command is the last access-actual shown in the list: in this case, the value 0.

- That evaluation of the structure-reference BLK2[2,4,12,0] results in the address value BLK2+8, since the component identified by the reference begins in the first byte of the third longword in segment BLK2. (More specifically, field F2 is defined as the 12 bits beginning with bit 4 of the third allocation unit.) As stated previously, the size and extension values are not involved in the evaluation.

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

If a structure reference appears in a "data" (righthand) operand of a DEPOSIT command, it has the same meaning as described above for the EVALUATE command, depending on whether or not a contents operator is specified. That is, with indirection specified the structure reference results in a fetch of the element or component indicated, with or without sign extension as requested. Without indirection, a structure reference results in the same kind of address evaluation as is performed by the EVALUATE command.

### 3.3.7 REF Structure References

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

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

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

### 3.3.8 Scope of Names

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

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

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

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

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

### 3.3.9 Source-Line Debugging

When the BLISS-32 compiler is executed under VAX/VMS Version 3 (or later versions), additional information is entered into the DEBUG Symbol Table, which permits a limited form of source-line number debugging (where the source-line numbers are defined as the numbers printed in the leftmost margin of a listing).

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

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

Note, however, that since BLISS-32 is an optimizing compiler for an expression language,all source lines cannot be accessed with the %LINE syntax.

However, using the TYPE command it is possible to display the source text for any line, including comments.

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

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

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

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

### 3.3.10 Effect of Compilation and Link-Time Qualifiers

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

The debugger is a shareable image. When a shareable image is linked into an executable image, it is unnecessary to copy the physical content of the shareable image. When a program that uses a shareable image is **run**, the copy from the shareable image file is run. In reality, debugger modules are mapped into memory; they are not physically there. (Refer to the VAX-11 Linker Reference Manual.)

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

Recall that, as described in Section 1.3.2.2, the qualifier /TRACEBACK is compilation default, while /DEBUG is not.

### 3.3.11 Debugger Command Summary

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

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

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

With the exception of ASCII character input, the debugger automatically converts lowercase input to uppercase. (That is, the debugger is not sensitive to the case of an input character.)

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

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

CTRL/"x" refers to the simultaneous typing of the CTRL key and the respective character key, that is, C, Y, or Z (refer to the VAX/VMS Command Language User's Guide for information on the complete list of CTRL functions). CTRL/"x" echoes at the terminal as ^x.

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

All command lines except CTRL functions must end with a carriage return.

@filespec

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

CALL name {(argument ,...)}

> Call routine by its symbolic name or by its virtual address (address expression is invalid) with optional argument list. An argument list must be enclosed by parentheses.

CANCEL ALL

Cancel all breakpoints, tracepoints, watchpoints, and user-set entry/display modes. Type is set to its default value (long integer), and scope is set to its default value, 0. The initial entry/display modes are restored. This command does not change the current contents of the debugger's symbol table (that is, those symbols acquired from program modules at debugger initialization or through use of the SET MODULE command, or any symbols that you defined during this debugging session). The current language is not changed.

CANCEL BREAK address-expression
CANCEL BREAK/ALL

Cancel breakpoint set at specified address, or cancel all breakpoints.

CANCEL EXCEPTION BREAK

Cancel the request that your program stop, as at a breakpoint, for any exception condition.

CANCEL MODE

Restore initial entry/display modes. Command does not change SCOPE or current language.

CANCEL MODULE module-name-list
CANCEL MODULE/ALL

Purge symbolic information associated with the named modules from the debugger's symbol table, or purge all module-related information from the symbol table. The typical use is to make space available for symbols associated with another module or modules (see SET MODULE). Global symbols and any symbols defined during this debugging session are not affected.

CANCEL SCOPE

Set scope to 0.

*CANCEL SOURCE{/module=modnam}

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

CANCEL TRACE address-expression
CANCEL TRACE/CALL
CANCEL TRACE/BRANCH
CANCEL TRACE/ALL

Cancel tracepoint set at specified address, cancel all opcode tracing at call-type instructions, cancel all opcode tracing at branch-type instructions, or cancel all tracepoints and opcode tracing.

CANCEL WATCH address-expression
CANCEL WATCH/ALL

Cancel watchpoint set at specified address, or cancel all watchpoints.

CTRL/C

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

CTRL/Y

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

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

CTRL/Z

Same result as EXIT; that is, terminate the debugging session and transfer control to the VAX/VMS command interpreter.

DEFINE symbol-name=value {,symbol-name=value ,...}

Equate name(s) of name=value list with associated value(s) for use during this debugging session. The debugger searches these symbols first whenever it requires a definition for a symbolic entry, and whenever it requires a symbolic name to report a location.

DEPOSIT{/modifier ...} address-expression=data{,data ,...}

Enter data specified in data list in sequence of locations beginning with the specified address. Modifier can be any mode or type keyword. (Mode and type keywords can be mixed in any meaningful way).

EVALUATE{/mode ...} expression ,...

Transform input (which can be field name, arithmetic expression, ASCII string, VAX-11 MACRO instruction, symbol, structure- or field-reference, or numeric value) to associated value(s) and display result(s). Can be used as desk calculator, radix converter, symbol verifier, and so on. The debugger displays result(s) in the order in which you specified the input. The displayed decimal value of the field name is a comma list of field components.

EXAMINE{/modifier ...} address{:address} {,address{:address} ,...}

Display current contents of specified address(es). The colon signifies range; that is, display contents of addresses from low address through high address. Modifier can be any mode or type keyword. (Mode and type keywords can be mixed in any meaningful way).

EXIT

Terminate debugging session and transfer control to the VAX/VMS command interpreter. An EXIT command within an indirect command file cancels one level of indirection; an EXIT command given at terminal input level terminates the debug process.

GO {address-expression}

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

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

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

*HELP topic {subtopic...}

Displays a description, format, qualifiers, and parameters of a debugger command as specified by the topic parameter; and describes qualifiers and/or parameters as specified by the subtopic parameter.

*SEARCH{/qualifier{/qualifier} } range string

Limits the debugger's search (of a source file) to specified program regions for occurrences of the string. The qualifiers are:

| | |
|---|---|
| /ALL | Specifies search for all occurrences of the string, and display of every line, in the specified range. |
| /NEXT | Specifies search, and display, of only the first occurrence of the string in the specified range. (This is the default.) |
| /IDENTIFIER | Specifies search for an occurrence of the string in the specified range. but the string is only displayed if it is bounded (on either side) by a character that is not part of an identifier in the current language. |
| /STRING | Specifies a search and display of an occurrence of the string specified. (This is the default.) |

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

| | |
|---|---|
| MODNAME | Search the specified module from line number 0 to end of module. |
| MODNAME\LINE-NUM | Search the specified module from the specified line number to the end of the module. |
| MODNAME\LINE-NUM:LINE-NUM | Search the specified module from the first line number given to the last. |
| LINE-NUM | Search the module designated by the current scope setting from the specified line number to the end of the module. |

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

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

SET BREAK address-expression {DO (command list)}

Establish breakpoint at specified address (the breakpoint stops your program before the instruction beginning with "address-expression" is executed).

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

You can specify the "after" option to defer a breakpoint:

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

Your program does not stop because of the breakpoint (that is, the breakpoint is ignored) until the "n"th pass through the specified location, as in an iteration, where n is within the range 1 through 32767. Thereafter, the breakpoint takes effect each time the debugger encounters it. You can specify a temporary (or one time) breakpoint by:

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

The debugger automatically cancels the breakpoint after it stops your program the first time the breakpoint is encountered.

SET EXCEPTION BREAK

Stop the program and report the current program counter contents if an exception condition occurs that was not initiated by a debugger command.

SET LANGUAGE language-name

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

SET LOG file-name

Specify a name other than the default name (DEBUG.LOG) for the debug log file. Can be used to generate multiple log files during a single debug session.

```
*SET MARGIN rm
*SET MARGIN lm:rm
*SET MARGIN lm:
*SET MARGIN   :rm
```

Specify the leftmost and/or rightmost source line character position at which to begin and end a line of source code. The default value for the left margin is 1; while the default value for the right margin is 255.

*SET MAX_SOURCE_FILES n

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

SET MODE mode-keyword{,mode-keyword ,...}

Allow or inhibit the entry and display of data in specified formats.

The following list describes the function of each keyword:

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

The debugger's initial modes are: SYMBOLIC and HEXADECIMAL.

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

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

SET MODULE module-name-list
SET MODULE/ALL

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

SET OUTPUT keyword{,keyword,...}

Turn the logging function on/off, display or inhibit the display of DEBUG commands executed from indirect command files or breakpoint actions, and permit or suppress DEBUG output, except error messages, to the terminal. Valid keywords are:

| | |
|---|---|
| LOG | Copy all command input (verbatim) and DEBUG output, including error messages, to the current log file; copy verified lines if VERIFY is specified. |

| | |
|---|---|
| NOLOG | Inhibit the creation of a debug log file. |
| TERMINAL | Print all debug output on the terminal, including command input, debug responses, comment lines, and warning and error messages. |
| NOTERMINAL | Print only error messages; suppress all debug output, including verified lines. |
| VERIFY | Echo command lines on the terminal as they are executed from a breakpoint action or indirect command file. |
| NOVERIFY | Do not echo command lines from indirect command file on the terminal. |

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

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

The following symbols may appear on the log file and/or terminal:

| | |
|---|---|
| ! | Indicates a DEBUG response |
| !! | Indicates a comment line |

To log all DEBUG command I/O on the log file, use the command:

```
SET OUTPUT LOG
```

SET SCOPE module-name{\routine-name ...}

Establish an ordered list of pathnames for use with the debugger's symbol search rules. A pathname completely and unambiguously identifies a symbol. For BLISS, a pathname is one of the following:

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

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

NOTE

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

- 0 - Indicates the pathname of the lexical entity, for example, routine or block, that contains the current program counter.
- 1, 2, 3,... - The pathname "1" indicates the caller of the lexical entity containing the current PC; pathname "2" indicates the caller of pathname "1", and so on.

- \ - The pathname "\" preceding a symbolic name indicates a global symbol of that name; for example:

  EXAM \GLOBAL

  causes a search for a global symbol whose name is 'GLOBAL'; nonglobal symbols of the same name are ignored.

(Refer to the VAX-11 Symbolic Debugger Reference Manual.)

*SET SEARCH parameter{,parameter}

Establishes search parameters whenever a SEARCH command qualifier is not specified. The parameters determine the search for occurrences of a string as follows: find all occurrences (ALL); find only the next occurrence (NEXT); display all occurrences found (STRING); display only the occurrences unbounded by a current language identifier (IDENTIFIER).

*SET SOURCE{/module=modname} dirname{,dirname...}

Directs the search of specified directories for source files. The command is used to locate source files that have been removed from compile-time directories.

SET STEP keyword {,keyword ,...}

Establish default conditions for the STEP command. Valid keywords are:

INSTRUCTION - Step increment in VAX-11 MACRO instructions.

INTO - Allow stepping through called routine.

LINE - Step increment by listing line numbers.

OVER - Step over called routine (make call transparent).

*SOURCE - Display the line of source code that corresponds to the instruction(s) being executed.

*NOSOURCE - Inhibit the display of the line of source code corresponding to the instruction being executed.

SYSTEM - Allow stepping in system space.

NOSYSTEM - Inhibit stepping in system space (make execution therein transparent).

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

SET TRACE address-expression
SET TRACE/CALL
SET TRACE/BRANCH

Set tracepoint at specified address, or specify tracing of all call-type instructions, or all branch-type instructions. At a tracepoint, the debugger reports the current program counter contents and then continues program execution automatically.

SET TYPE

Specify a data type to be associated to data when DEBUG cannot infer a type from its input.

The following list describes the function of each keyword:

| | |
|---|---|
| ASCII:n | Interpret/display data as a string of n ASCII characters, where n is an integer. |
| BYTE | Interpret/display data in byte units. |
| INSTRUCTION | Interpret/display VAX-11 MACRO instructions. |
| LONG | Interpret/display data in longword units. |
| WORD | Interpret/display data in word units. |

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

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

SET WATCH address-expression

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

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

SHOW BREAK

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

SHOW CALLS {n}

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

SHOW LOG

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

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

*SHOW MARGIN

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

*SHOW MAX_SOURCE_FILES

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

SHOW MODE

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

SHOW MODULE

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

SHOW OUTPUT

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

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

*SHOW SEARCH

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

SHOW SCOPE

Report the current contents of SCOPE.

*SHOW SOURCE

Displays the current source directory search list(s) established by the SET SOURCE or SET SOURCE/module=modname command.

SHOW STEP

Report current default conditions for STEP (see SET STEP).

SHOW TRACE

Report the locations of current tracepoints, or that opcode tracing is in effect.

SHOW TYPE

Report the current type setting.

SHOW WATCH

Report the locations of current watchpoints and the number of bytes monitored by each watchpoint.

STEP{/keyword} {decimal-integer}

Begin program execution and then stop after executing the specified number of instructions or listing-lines. (If you do not specify a count, the value 1 is assumed; that is, single stepping is the default.) The count is a decimal integer between 0 through 32767.

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

The keywords have the following relationships:

SYSTEM/NOSYSTEM - Count/do not count steps in system space.

INTO/OVER - Count/do not count steps within a called routine.

INSTRUCTION - Step by instructions.

LINE - Step by listing line numbers.

*SOURCE/NOSOURCE - Display/do not display line(s) of source code.

The initialized defaults for BLISS are:

INSTRUCTION, NOSYSTEM, OVER.

```
*TYPE { {modname\}line-number{:line-number} -
       {,{modname\}line-number{:line-number}...} }
```

Displays the source language statement(s) corresponding to the line number(s) specified.

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

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

# CHAPTER 4

# MACHINE-SPECIFIC FUNCTIONS

Machine-specific functions (also referenced as builtin functions) allow you to perform specialized VAX-11 operations within the BLISS language. A machine-specific function call is similar to a BLISS routine call. It requires parameters and, in some cases, returns a value. If a machine-specific function that does not return a value is used in a context that requires a value, an error is reported, as in a BLISS routine.

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

Machine-specific functions in BLISS-32 are divided into categories, as illustrated in Table 4-1. A separate description of each function is given in the following sections. For a detailed discussion, consult the VAX-11/780 Architecture Handbook.

The definitions of these functions consistently require addresses as parameters, even where values could have been specified for what VAX-11 terms "source operands" that are a longword or less in size. For example, where a length is required as the second parameter of the PROBER function, the call is written as in:

```
PROBER(...,UPLIT(5),...)
```

or

```
PROBER(...,%REF(.A+2),...)
```

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

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

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

is equivalent to:

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

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

**Rule 1** - A %REF value cannot be used with a DESTINATION address.

**Rule 2** - An undotted register can only be used for operands that are: BYTE, WORD, LONG, or Single-precision-floating-point.

**Rule 3** - A register name cannot be used as the address of a character string or packed decimal string.

**Table 4-1: Machine-Specific Functions**

| | |
|---|---|
| **Processor Register Operations** | |
| MFPR | Move From Processor Register |
| MTPR | Move to Processor Register |
| **Parameter Validation Operations** | |
| PROBER | Probe Read Accessibility |
| PROBEW | Probe Write Accessibility |
| **Program Status Operations** | |
| BICPSW | Bit Clear PSW |
| BISPSW | Bit Set PSW |
| MOVPSL | Move From PSL |
| **Queue Operations** | |
| INSQUE | Insert Entry Into Queue |
| INSQxI | Insert Entry Into Queue Interlocked |
| REMQUE | Remove Entry From Queue |
| REMQxI | Remove Entry From Queue Interlocked |
| **Bit Operations** | |
| FFC | Find First Clear Bit |
| FFS | Find First Set Bit |
| TESTBITCC | Test for Bit Clear; Clear Bit |
| TESTBITCCI | Test for Bit Clear; Clear Bit Interlocked |
| TESTBITCS | Test for Bit Clear; Set Bit |
| TESTBITSC | Test for Bit Set; Clear Bit |
| TESTBITSS | Test for Bit Set; Set Bit |
| TESTBITSSI | Test for Bit Set; Set Bit Interlocked |

(continued on next page)

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

| | |
|---|---|
| **Arithmetic Operations** | |
| ADAWI | Add Aligned Word Interlocked |
| ADDD | Add Double Operands |
| ADDF | Add Float operands |
| ADDG | Add Float-G Operands |
| ADDH | Add Float-H Operands |
| ADDM | Add Multiword Operands |
| ASHQ | Arithmetic Shift Quad |
| DIVD | Divide Double Operands |
| DIVF | Divide Float Operands |
| DIVG | Divide Float-G Operands |
| DIVH | Divide Float-H Operands |
| EDIV | Extended-Precision Divide |
| EMUL | Extended-Precision Multiply |
| MULD | Multiply Double Operands |
| MULF | Multiply Floating Operands |
| MULG | Multiply Float-G Operands |
| MULH | Multiply Float-H Operands |
| SUBD | Subtract Double Operands |
| SUBF | Subtract Floating Operands |
| SUBG | Subtract Float-G Operands |
| SUBH | Subtract Float-H Operands |
| SUBM | Subtract Multiword Operands |
| **Arithmetic Comparison Operations** | |
| CMPD | Compare Double Operands |
| CMPF | Compare Float Operands |
| CMPG | Compare Float-G Operands |
| CMPH | Compare Float-H Operands |
| CMPM | Compare Multiword Operands |
| **Arithmetic Conversion Operations** | |
| CVTDF | Convert Double to Float |
| CVTDI | Convert Double to Integer |
| CVTDL | Convert Double to Long |
| CVTFD | Convert Float to Double |
| CVTFG | Convert Float to Float-G |
| CVTFH | Convert Float to Float-H |
| CVTFI | Convert Float to Integer |
| CVTFL | Convert Float to Long |
| CVTGF | Convert Float-G to Float |
| CVTGL | Convert Float-G to Long |
| CVTHF | Convert Float-H to Float |
| CVTHL | Convert Float-H to Long |
| CVTID | Convert Integer to Double |
| CVTIF | Convert Integer to Float |
| CVTLD | Convert Long to Double |
| CVTLF | Convert Long to Floating |
| CVTLH | Convert Long to Float-H |
| CVTRDH | Convert Rounded Double to Float-H |
| CVTRDL | Convert Rounded Double to Long |
| CVTRFL | Convert Rounded Float to Long |
| CVTRGH | Convert Rounded Float-G to Float-H |
| CVTRGL | Convert Rounded Float-G to Long |
| CVTRHL | Convert Rounded Float-H to long |

(continued on next page)

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

| | |
|---|---|
| **Character String Operations** | |
| CMPC3 | Compare Characters 3 Operand |
| CMPC5 | Compare Characters 5 Operand |
| CRC | Cyclic Redundancy Calculation |
| LOCC | Locate Character |
| MATCHC | Match Characters |
| MOVC3 | Move Character 3 Operand |
| MOVC5 | Move Character 5 Operand |
| MOVTC | Move Translated Characters |
| MOVTUC | Move Translated Until Character |
| SCANC | Scan Characters |
| SKPC | Skip Character |
| SPANC | Span Characters |
| **Decimal String Operations** | |
| ASHP | Arithmetic Shift and Round Packed |
| CMPP | Compare Packed |
| CVTLP | Convert Long to Packed |
| CVTPL | Convert Packed to Long |
| CVTPS | Convert Packed to Leading Separate Numeric |
| CVTPT | Convert Packed to Trailing Numeric |
| CVTSP | Convert Leading Separate Numeric to Packed |
| CVTTP | Convert Trailing Numeric to Packed |
| EDITPC | Edit Packed to Character |
| MOVP | Move Packed |
| **Miscellaneous Operations** | |
| BPT | Breakpoint |
| BUGL | Bugcheck With Long Operand |
| BUGW | Bugcheck With Word Operand |
| CALLG | Call With General Argument List |
| CHMx | Change Mode |
| HALT | Halt Processor |
| INDEX | Index (Subscript) Calculation |
| NOP | No Operation |
| ROT | Rotate a Value |
| XFC | Extended Function Call |

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

BLISS Source

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

Generated Code

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

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

```
        BBSS    #3,X,1$
        BRW     2$
1$:
        ...             ;
        ...             ; CONSEQUENCE
        ...             ;
2$:
```

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

BLISS Source

```
        Y = TESTBITSS( X<3,1> )
```

Generated Code

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

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

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

would be coded as:

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

without any temporary being allocated.

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

```
        MOVPSL(ALOCAL)
```

might result in

```
        MOVPSL  R2
```

## 4.1 ADAWI - ADD ALIGNED WORD INTERLOCKED

ADAWI (SRCADDR, DSTADDR)

Parameters:

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

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

Result:

Contents of the PSL

## 4.2 ADDD - ADD DOUBLE OPERANDS

ADDD (SRC1A, SRC2A, DSTA)

Parameters:

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

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

DSTA - Address of a quadword where the sum is stored

Result:

NOVALUE

## 4.3 ADDF - ADD FLOATING OPERANDS

ADDF (SRC1A, SRC2A, DSTA)

Parameters:

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

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

DSTA - Address of a longword where the sum is stored

Result:

NOVALUE

## 4.4 ADDG - ADD FLOAT-G OPERANDS

ADDG (SRC1A, SRC2A, DSTA)

Parameters:

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

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

DSTA - Address of a quadword where the sum is stored

Result:

NOVALUE

## 4.5 ADDH - ADD FLOAT-H OPERANDS

ADDH (SRC1A, SRC2A, DSTA)

Parameters:

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

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

DSTA - Address of an octaword where the sum is stored

Results:

NOVALUE

## 4.6 ADDM - ADD MULTIWORD OPERANDS

ADDM (SIZE, SRC1A, SRC2A, DSTA)

Parameters:

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

SRC1A - Address of extended multi-precision integer used as the addend

SRC2A - Address of extended multi-precision integer used as the augend

DSTA - Address of the sum of source one and source two

Result:

NOVALUE

## 4.7 ASHQ - ARITHMETIC SHIFT QUAD

ASHQ (SHIFT, SRCADDR, DSTADDR)

Parameters:

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

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

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

Result:

Contents of the PSL

## 4.8 BICPSW - BIT CLEAR PSW

BICPSW (MASKADDR)

Parameter:

MASKADDR - Address of a word whose contents are to be ones complemented and AND'ed into the PSW

Result:

NOVALUE

## 4.9 BISPSW - BIT SET PSW

BISPSW (MASKADDR)

Parameter:

MASKADDR - Address of a word whose contents are to be OR'ed into the PSW

Result:

NOVALUE

## 4.10 BPT - BREAK POINT TRAP

BPT ()

Parameters:

None

Result:

NOVALUE

## 4.11 BUGL - BUGCHECK WITH LONG OPERAND

BUGL (ARG)

Parameters:

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

Result:

NOVALUE

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

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

generates the code sequence:

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

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

## 4.12 BUGW - BUGCHECK WITH WORD OPERAND

BUGW (ARG)

Parameters:

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

Result:

NOVALUE

## 4.13 CALLG - CALL WITH GENERAL PARAMETER LIST

CALLG (ARGLIST, RTN)

Parameters:

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

RTN - Address of the routine to be called

Result:

Same as result of the routine that is called

Note:

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

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

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

## 4.14 CHMX - CHANGE MODE

```
CHME (ARG)
CHMK (ARG)
CHMS (ARG)
CHMU (ARG)
```

Parameters:

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

Result:

NOVALUE

## 4.15 CMPD - COMPARE DOUBLE

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

Parameters:

SRC1A - Address of a quadword containing a double-precision floating-point value be the name of a register nor a %REF value)

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

Result:

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

## 4.16 CMPF - COMPARE FLOATING

CMPF (SRC1A, SRC2A)

Parameters:

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

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

Result:

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

## 4.17 CMPP - COMPARE PACKED

CMPP (SRC1LENA, SRC1ADDR, SRC2LENA, SRC2ADDR)

Parameters:

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

SRC1ADDR - Address of the base of packed decimal string SRC1

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

SRC2ADDR - Address of the base of packed decimal string SRC2

Result:

-1 - SRC1 less than SRC2
0 - SRC1 equal to SRC2
1 - SRC1 greater than SRC2

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

## 4.18 CRC - CYCLIC REDUNDANCY CHECK

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

Parameters:

TABLEADDR - Address of a 16-longword table

INICRCADDR - Address of a longword which contains the initial CRC

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

STREAMADDR - Address of the first byte of the data stream

DSTADDR - Address of a longword where the resulting 32-bit CRC is to be stored

Result:

NOVALUE

## 4.19 CVTDF - CONVERT DOUBLE TO FLOATING

CVTDF (SRCA, DSTA)

Parameters:

SRCA - Address of a quadword containing a double-precision floating-point value

DSTA - Address of a longword where the single-precision floating-point conversion is stored

Result:

1 - No floating-point overflow

0 - Floating-point overflow

## 4.20 CVTDI - CONVERT DOUBLE TO INTEGER

CVTDI (SRCA, DSTA)

Parameters:

SRCA - Address of a quadword containing a double-precision floating-point value

DSTA - Address where the integer conversion is stored

Result:

1 - No integer overflow

0 - integer overflow

## 4.21 CVTDL - CONVERT DOUBLE TO LONG

CVTDL (SRCA, DSTA)

Parameters:

SRCA - Address of a quadword containing a double-precision floating-point value

DSTA - Address of a longword where the integer conversion is stored

Result:

1 - No integer overflow

0 - Integer overflow

## 4.22 CVTFD - CONVERT FLOATING TO DOUBLE

CVTFD (SRCA, DSTA)

Parameters:

SRCA - Address of a longword containing a single-precision floating-point value

DSTA - Address of a quadword where the double-precision floating-point conversion is stored

Result:

NOVALUE

## 4.23 CVTFG - CONVERT FLOATING TO FLOAT-G

CVTFG (SRCA, DSTA)

Parameters:

SRCA - Address of a longword containing a single-precision floating-point value

DSTA - Address of a quadword where the extended double-precision floating-point conversion is stored

Result:

NOVALUE

## 4.24 CVTFH - CONVERT FLOATING TO FLOAT-H

CVTFH (SRCA, DSTA)

Parameters:

SRCA - Address of a longword containing a single-precision floating-point value

DSTA - Address of an octaword where the extended-exponent double-precision floating-point conversion is stored

Result:

NOVALUE

## 4.25 CVTFI - CONVERT FLOATING TO INTEGER

CVTFI (SRCA, DSTA)

Parameters:

SRCA - Address of a longword containing a single-precision floating-point value

DSTA - Address of a longword where the integer conversion is stored

Result:

1 - No integer overflow

0 - Integer overflow

## 4.26 CVTFL - CONVERT FLOATING TO LONG

CVTFL (SRCA, DSTA)

Parameters:

SRCA - Address of a longword containing a single-precision floating-point value

DSTA - Address of a longword where the integer conversion is stored

Result:

1 - No integer overflow

0 - Integer overflow

## 4.27 CVTGF - CONVERT FLOAT-G TO FLOATING

CVTGF (SRCA, DSTA)

Parameters:

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

DSTA - Address of a longword where the single-precision floating-point conversion is stored

Result:

NOVALUE

## 4.28 CVTGL - CONVERT FLOAT-G TO LONG

CVTGL (SRCA, DSTA)

Parameters:

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

DSTA - Address of a longword where the integer conversion is stored

Result:

NOVALUE

## 4.29 CVTHF - CONVERT FLOAT-H TO FLOATING

CVTHF (SRCA,DSTA)

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

DSTA - Address of a longword where the single-precision floating-point conversion is stored

Result:

NOVALUE

## 4.30 CVTHL - CONVERT FLOAT-H TO LONG

CVTHL (SRCA, DSTA)

Parameters:

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

DSTA - Address of a longword where the integer conversion is stored

Result:

NOVALUE

## 4.31 CVTID - CONVERT INTEGER TO DOUBLE

CVTID (SRCA, DSTA)

Parameters:

SRCA - Address of a longword containing an integer value

DSTA - Address of a quadword where the double-precision floating-point conversion is stored

Result:

NOVALUE

## 4.32 CVTIF - CONVERT INTEGER TO FLOATING

CVTIF (SRCA, DSTA)

Parameters:

SRCA - Address of a longword containing an integer value

DSTA - Address of a longword where the single-precision floating point conversion is stored

Result:

NOVALUE

## 4.33 CVTLD - CONVERT LONG TO DOUBLE

CVTLD (SRCA, DSTA)

Parameters:

SRCA - Address of a longword containing an integer value

DSTA - Address of quadword where the double-precision floating-point conversion is stored

Result:

NOVALUE - No overflow can occur

## 4.34 CVTLF - CONVERT LONG TO FLOATING

CVTLF (SRCA, DSTA)

Parameters:

SRCA - Address of a longword containing an integer value

DSTA - Address of a longword where the single-precision floating-point conversion is stored

Result:

NOVALUE - No overflow can occur

## 4.35 CVTLH - CONVERT LONG TO FLOAT-H

CVTLH (SRCA, DSTA)

Parameters:

SRCA - Address of a longword containing an integer value

DSTA - Address of an octaword where the extended-exponent double-precision floating point conversion is stored

Result:

NOVALUE

## 4.36 CVTLP - CONVERT LONG TO PACKED

CVTLP (SRCA, DSTLENA, DSTADDR)

Parameters:

| | |
|---|---|
| SRCA | - Address of a longword containing an integer value |
| DSTLENA | - Address of a word containing the length of the destination string |
| DSTADDR | - Address of the base of the destination string |

Result:

| | |
|---|---|
| 1 | - No decimal overflow |
| 0 | - Decimal overflow |

## 4.37 CVTPL - CONVERT PACKED TO LONG

CVTPL (SRCLENA, SRCADDR, DSTA)

Parameters:

| | |
|---|---|
| SRCLENA | - Address of a word containing the length of the source string |
| SRCADDR | - Address of the base of the source string |
| DSTA | - Address of a longword where the integer conversion is stored |

Result:

| | |
|---|---|
| 1 | - No integer overflow |
| 0 | - Integer overflow |

## 4.38 CVTPS - CONVERT PACKED TO LEADING SEPARATE NUMERIC

CVTPS (SRCLENA, SRCADDR, DSTLENA, DSTADDR)

Parameters:

| | |
|---|---|
| SRCLENA | - Address of a word containing the length of the source string |
| SRCADDR | - Address of the base of the source string |
| DSTLENA | - Address of a word containing the length of the destination string |
| DSTADDR | - Address of the base of the destination string |

Result:

| | |
|---|---|
| 1 | - No decimal overflow |
| 0 | - Decimal overflow |

## 4.39 CVTPT - CONVERT PACKED TO TRAILING NUMERIC

CVTPT (SRCLENA, SRCADDR, TBLADDR, DSTLENA, DSTADDR)

Parameters:

SRCLENA - Address of a word containing the length of the source string

SRCADDR - Address of the base of the source string

TBLADDR - Address of the table used to convert the sign

DSTLENA - Address of a word containing the length of the destination string

DSTADDR - Address of the base of the destination string

Result:

1 - No decimal overflow

0 - Decimal overflow

## 4.40 CVTRDH - CONVERT ROUNDED DOUBLE TO FLOAT-H

CVTRDH (SRCA, DSTA)

Parameters:

SRCA - Address of a quadword containing a double-precision floating-point value

DSTA - Address of an octaword where the extended-exponent double-precision floating-point conversion is stored

Result:

NOVALUE

## 4.41 CVTRDL - CONVERT ROUNDED DOUBLE TO LONG

CVTRDL (SRCA, DSTA)

Parameters:

SRCA - Address of a quadword containing a double-precision floating-point value

DSTA - Address of a longword where the integer conversion is stored

Result:

1 - No integer overflow

0 - Integer overflow

## 4.42 CVTRFL - CONVERT ROUNDED FLOATING TO LONG

CVTRFL (SRCA, DSTA)

Parameters:

SRCA - Address of a longword containing a single-precision floating-point value

DSTA - Address of a longword where the integer conversion is stored

Result:

1 - No integer overflow

0 - Integer overflow

## 4.43 CVTSP - CONVERT LEADING SEPARATE TO PACKED

CVTSP (SRCLENA, SRCADDR, DSTLENA, DSTADDR)

Parameters:

SRCLENA - Address of a word containing the length of the source string

SRCADDR - Address of the base of the source string

DSTLENA - Address of a word containing the length of the destination string

DSTADDR - Address of the base of the destination string

Result:

1 - No decimal overflow

0 - Decimal overflow

## 4.44 VTTP - CONVERT TRAILING NUMERIC TO PACKED

CVTTP (SRCLENA, SRCADDR, TBLADDR, DSTLENA, DSTADDR)

Parameters:

SRCLENA - Address of a word containing the length of the source string

SRCADDR - Address of the base of the source string

TBLADDR - Address of the table used to convert the sign

DSTLENA - Address of a word containing the length of the destination string

DSTADDR - Address of the base of the destination string

Result:

1 - No decimal overflow

0 - Decimal overflow

## 4.45 DIVD - DIVIDE DOUBLE OPERANDS

DIVD (DIVSR, DIVID, QUOT)

Parameters:

DIVSR - Address of a double-precision floating-point quadward used as the divisor

DIVID - Address of a double-precision floating-point quadword used as the dividend

QUOT - Address of a quadword where the quotient is stored

Result:

NOVALUE

## 4.46 DIVF - DIVIDE FLOATING OPERANDS

DIVF (DVISR DIVID, QUOT)

DVISR - Address of a single-precision floating-point longword used as the divisor

DIVID - Address of a single-precision floating-point longword used as the dividend

QUOT - Address of a longword where the quotient is stored

Result:

NOVALUE

## 4.47 DIVG - DIVIDE FLOAT-G OPERANDS

DIVG (DIVSR, DIVID, QUOT)

Parameters:

DIVSR - Address of an extended double-precision floating point quadword used as the divisor

DIVID - Address of an extended double-precision floating point quadword used as the dividend

QUOT - Address of a quadword where the quotient is stored

Result:

NOVALUE

## 4.48 DIVH - DIVIDE FLOAT-H OPERANDS

DIVH (DIVSR, DIVID, QUOT)

Parameters:

| | |
|---|---|
| DIVSR | - Address of an extended-exponent double-precision floating-point octaword that is used as the divisor |
| DIVID | - Address of an extended-exponent doublw-precision floating-point octaword that is used as the dividend |
| QUOT | - Address of an octaword where the quotient is stored |

Result:

NOVALUE

## 4.49 EDITPC - EDIT PACKED TO CHARACTER

EDITPC (SRCLENA, SRCADDR, PATTERN, DSTADDR)

Parameters:

| | |
|---|---|
| SRCLENA | - Address of a word containing the length of the source string |
| SRCADDR | - Address of the base of the source string |
| PATTERN | - Address of the pattern-operator string |
| DSTADDR | - Address of the base of the destination string |

Result:

| | |
|---|---|
| 1 | - No decimal overflow |
| 0 | - Decimal overflow |

## 4.50 EDIV - EXTENDED-PRECISION DIVIDE

EDIV (DIVISOR, DIVIDEND, QUOTIENT, REMAINDER)

Parameters:

| | |
|---|---|
| DIVISOR | - Address of a longword whose contents are used as the divisor |
| DIVIDEND | - Address of a quadword whose contents are used as the dividend |
| QUOTIENT | - Address of a longword where the quotient is to be stored |
| REMAINDER | - Address of a longword where the remainder is to be stored |

Result:

Contents of the PSL

## 4.51 EMUL - EXTENDED-PRECISION MULTIPLY

EMUL (MULR, MULD, ADD, PROD)

Parameters:

MULR - Address of a longword whose contents are used as the multiplier

MULD - Address of a longword whose contents are used as the multiplicand

ADD - Address of a longword whose contents are sign-extended to a quadword and added to the product quadword

PROD - Address of a quadword where the result is to be stored a %REF value)

Result:

Contents of the PSL

## 4.52 FFC AND FFS - FIND AND MODIFY OPERATIONS

FFC (POSADDR, SIZADDR, BASEADDR, DSTADDR) Find first clear bit
FFS (POSADDR, SIZADDR, BASEADDR, DSTADDR) Find first set bit

Parameters:

POSADDR - Address of a longword whose contents specify a bit position relative to the low bit of the byte addressed by BASEADDR. The low bit of that byte is bit position zero.

SIZADDR - Address of a byte whose contents, when zero extended to 32 bits, have a value less than or equal to 32. This value specifies the size of the field to be searched. The size of the field is measured in bits and begins at the bit position specified by POSADDR and extends toward increasing bit numbers.

BASEADDR - Address of a byte whose low bit is interpreted as position zero

DSTADDR - Address of a longword where the position of the first bit found in the specified state is to be stored

Result:

1 - No bit in the specified state is found in the field searched.

0 - Otherwise

## 4.53 HALT - HALT PROCESSOR

HALT ()

Parameters:

NONE

Result:

NOVALUE

Description:

This routine generates a HALT instruction.

## 4.54 INDEX - INDEX CALCULATION

INDEX (SUBSCRIPT, LOW, HIGH, SIZE, INDEXIN, INDEXOUT)

Parameters:

| | |
|---|---|
| SUBSCRIPT | - Address of a longword containing the subscript |
| LOW | - Address of a longword containing the lower bound of the subscript range |
| HIGH | - Address of a longword containing the upper bound of the subscript range |
| SIZE | - Address of a longword containing the scale factor |
| INDEXIN | - Address of a longword containing the initial index value |
| INDEXOUT | - Address of a longword where the result is to be stored |

Result:

NOVALUE

## 4.55 INSQHI AND INSQTI - INSERT ENTRY IN QUEUE, INTERLOCKED

INSQHI(ENTRY, HEADER) Insert entry in queue at head, interlocked
INSQTI(ENTRY, HEADER) Insert entry in queue at tail, interlocked

Parameters:

| | |
|---|---|
| ENTRY | - Address of an entry to be inserted in a queue following the header |
| HEADER | - Address of the queue header |

Result:

| | |
|---|---|
| 0 | - If ENTRY was not the first entry to be inserted in the queue |
| 1 | - If the secondary interlock could not be acquired |
| 2 | - If ENTRY was the first entry to be inserted into the queue |

Note that, if a real result is needed, the instruction sequence generated for these functions is:

```
        CLRL    tmp
        INSQxI  ENTRY,HEADER
        BCS     1$
        BNEQ    2$
        INCL    tmp
1$:     INCL    tmp
2$:
```

If only a flow result is requested, the instruction sequence is:

```
        INSQxI  ENTRY,HEADER
        BCC     1$              ; Success
        ....                    ; Failure actions
```

Thus, the BLISS expression to set a software interlock realized with a queue is:

```
WHILE INSQHI(ENTRY, HEADER ) DO WAIT();
```

## 4.56 INSQUE - INSERT ENTRY IN QUEUE

INSQUE (ENTRY, PRED)

Parameters:

| | |
|---|---|
| ENTRY | - Address of an entry to be inserted in the queue after the entry specified by PRED |
| PRED | - Address of an entry in a queue |

Result:

| | |
|---|---|
| 1 | - ENTRY was the first entry to be inserted into the queue |
| 0 | - Otherwise |

## 4.57 LOCC - LOCATE CHARACTER

LOCC (CHARA, LENA, ADDR)

Parameters:

| | |
|---|---|
| CHARA | - Address of a byte containing the character to be located |

LENA - Address of a word containing the search string length

ADDR - Address of the string to be searched

Result:

TRUE - A byte was located which was the same as .CHARA<0,8> (the Z condition code is clear)

FALSE - No byte was located (the Z condition code is set)

## 4.58 MATCHC - MATCH CHARACTERS

MATCHC (OBJLENA, OBJADDR, SRCLENA, SRCADDR)

Parameters:

OBJLENA - Address of word containing the length of the pattern string

OBJADDR - Address of the pattern string

SRCLENA - Address of a word containing the length of the string to be searched

SRCADDR - Address of the string to be searched

Result:

TRUE - A match was found (the Z condition code is set)

FALSE - No match was found (the Z condition code is clear)

## 4.59 MFPR - MOVE FROM PROCESSOR REGISTER

MFPR (PROCREG, DSTADDR)

Parameters:

PROCREG - Processor register number

DSTADDR - Address of a longword where the contents of the processor register is to be stored

Result:

NOVALUE

## 4.60 MOVC3 - MOVE CHARACTER 3 OPERAND

MOVC3 (LENA, SRCADDR, DSTADDR)

Parameters:

LENA - Address of a word containing the length of the source string

SRCADDR - Address of the base of the source string

DSTADDR - Address of the destination string

Result:

NOVALUE

## 4.61 MOVC5 - MOVE CHARACTER 5 OPERAND

MOVC5 (SRCLENA, SRCADDR, FILL,TBLADDR, DSTLENA, DSTADDR)

Parameters:

SRCLENA - Address of word containing the length of the source string

SRCADDR - Address of the base of the source string

FILL - Address of a byte containing the fill character

TBLADDR - Address of the translation table

DSTLENA - Address of word containing the destination-string length

DSTADDR - Address of the destination string

Result:

NOVALUE

## 4.62 MOVP - MOVE PACKED

MOVP (LENA, SRCADDR, DSTADDR)

Parameters:

LENA - Address of a word containing the length of the source string

SRCADDR - Address of the base of the source decimal string

DSTADDR - Address of the base of the destination

Result:

NOVALUE

## 4.63 MOVPSL - MOVE FROM PSL

MOVPSL (DSTADDR)

Parameter:

DSTADDR - Address of a longword where the contents of the PSL is to be stored

Result:

NOVALUE

## 4.64 MOVTC - MOVE TRANSLATED CHARACTERS

MOVTC (SRCLENA, SRCADDR, FILL, TBLADDR, DSTLENA, DSTADDR)

Parameters:

| | |
|---|---|
| SRCLENA | - Address of a word containing the source length |
| SRCADDR | - Address of source string |
| FILL | - Address of a byte containing the fill character |
| TBLADDR | - Address of the translation table |
| DSTLENA | - Address of a word containing the destination length |
| DSTADDR | - Address of the destination string |

Result:

NOVALUE

## 4.65 MOVTUC - MOVE TRANSLATED UNTIL CHARACTER

MOVTUC (SRCLENA, SRCADDR, ESCA, TBLADDR, DSTLENA, DSTADDR)

Parameters:

| | |
|---|---|
| SRCLENA | - Address of a word containing the length of the source string |
| SRCADDR | - Address of the base of the source string |
| ESCA | - Address of a byte containing the escape character |
| TBLADDR | - Address of the translation table (may be created using CH$TRANSTABLE) |
| DSTLENA | - Address of a word containing the length of the destination string |
| DSTADDR | - Address of the base of the destination string |

Result:

If the string was successfully translated without escape, the result is 0. Otherwise, the result is the address of the byte in the source string which caused the escape.

## 4.66 MTPR - MOVE TO PROCESSOR REGISTER

MTPR (SRCADDR, PROCREG)

Parameters:

SRCADDR - Address of a longword whose contents are to be loaded into the designated processor register

PROCREG - Processor register number

Result:

NOVALUE

## 4.67 MULD - MULTIPLY DOUBLE OPERANDS

MULD (SRC1A, SRC2A, DSTA)

Parameters:

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

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

DSTA - Address of a quadword where the product is stored

Result:

NOVALUE

## 4.68 MULF - MULTIPLY FLOATING OPERANDS

MULF (SRC1A, SRC2A, DSTA)

Parameters:

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

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

DSTA - Address of a longword where the product is stored

Result:

NOVALUE

## 4.69 MULG - MULTIPLY FLOAT-G OPERANDS

MULG (SRC1A, SRC2A, DSTA)

Parameters:

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

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

DSTA - Address of a quadword where the product is stored

Result:

NOVALUE

## 4.70 MULH - MULTIPLY FLOAT-H OPERANDS

MULH (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A - Address of an extended-exponent double-precision floating-point octaword used as a multiplier

SRC2A - Address of an extended-exponent double-precision floating-point octaword used as a multiplicand

DSTA - Address of an octaword where the product is stored

Result:

NOVALUE

## 4.71 NOP - NO OPERATION

NOP ()

Parameters:

None

Result:

NOVALUE

## 4.72 PROBER - PROBE READ ACCESSIBILITY

PROBER (MODEADDR, LENGTHADDR, BASEADDR)

Parameters:

MODEADDR - Address of a byte whose contents specify a protection mode

LENGTHADDR - Address of a word whose contents are zero-extended and added to BASEADDR to specify the last byte of an area in memory

BASEADDR - Address of the first byte of an area in memory (must not be the name of a register nor a %REF value)

Result:

1 - Both first and last bytes are read-accessible

0 - Otherwise

## 4.73 PROBEW - PROBE WRITE ACCESSIBILITY

PROBEW (MODEADDR, LENGTHADDR, BASEADDR)

Parameters:

MODEADDR - Address of a byte whose contents specify a protection mode

LENGTHADDR - Address of a word whose contents are zero-extended and added to BASEADDR to specify the last byte of an area in memory

BASEADDR - Address of the first byte of an area in memory (must not be the name of a register nor a %REF value)

Result:

1 - Both first and last bytes are write-accessible

0 - Otherwise

## 4.74 REMQHI AND REMQTI - REMOVE ENTRY FROM QUEUE, INTERLOCKED

REMQHI (HEADER, ADDR) Remove from queue at head, interlocked
REMQTI (HEADER, ADDR) Remove from queue at tail, interlocked

Parameters:

HEADER - Address of the queue header

ADDR - Address of a longword where the address of the entry removed is to be stored

Result:

0 - The queue is not empty; an entry was removed.

1 - The secondary interlock could not be acquired.

2 - The queue is now empty; the last entry was removed.

3 - The queue was already empty; no entry was removed.

The result of REMQxI is TRUE if no entry was removed, either because the queue was empty or because the secondary interlock was not acquired.

The instruction sequence generated in real-context is:

```
        CLRL    tmp
        REMQxI  HEADER,ADDR
        BCS     2$              ; Interlock failed
        BVS     1$              ; Remove failed
        BEQL    3$              ; Removed an entry
        DECL    tmp             ; Removed last entry
1$:     ADDL    #2,tmp
2$:     INCL    tmp
```

The instruction sequence generated in flow-context is:

```
        REMQxI  HEADER,ADDR
        BVC     1$              ; Entry removed
        .....                   ; Remove failed
```

## 4.75 REMQUE - REMOVE ENTRY FROM QUEUE

REMQUE (ENTRY, ADDR)

Parameters:

ENTRY - Address of an entry in a queue

ADDR - Address of a longword where the address of the entry removed is to be stored

Result:

| Value | Initial State | Final State | Entry Removed |
|---|---|---|---|
| 0 | not empty | not empty | yes |
| 2 | not empty | empty | yes |
| 3 | empty | empty | no |

Note that the value of REMQUE is true only if no entry was removed.

The value is computed from the condition codes as:

(Z-bit)^1 OR (V-bit)

## 4.76 ROT - ROTATE A VALUE

ROT (VALUE, SHIFT)

Parameters:

VALUE - Value to be rotated

SHIFT - Number of bits to rotate

Result:

VALUE rotated the specified number of bits

## 4.77 SCANC - SCAN CHARACTERS

SCANC (LENA, ADDR, TBLADDR, MASKA)

Parameters:

LENA - Address of a word containing the length of the string to be scanned

ADDR - Address of the base of the string

TBLADDR - Address of the translation table

MASKA - Address of a byte containing the mask to use in scanning

Result:

If the SCANC fails to find a match, the result is 0. Otherwise, the result is the address of the byte that produced a nonzero AND with the MASK.

## 4.78 SKPC - SKIP CHARACTER

SKPC (CHARA, LENA, ADDR)

Parameters:

CHARA - Address of a byte containing the character to be skipped

LENA - Address of a word containing the search string length

ADDR - Address of the string to be searched

Result:

TRUE - A byte was located which was not the same as .CHARA<0,8> (the Z condition code is clear)

FALSE - No byte was located (the Z condition code is set)

## 4.79 SPANC - SPAN CHARACTERS

SPANC (LENA, ADDR, TBLADDR, MASKA)

Parameters:

LENA - Address of a word containing the length of the string to be spanned

ADDR - Address of the base of the string

TBLADDR - Address of the translation table

MASKA - Address of a byte containing the mask to use in spanning

Result:

If the SPANC fails to find a match, the result is 0. Otherwise, the result is the address of the byte that produced a zero AND with the MASK.

## 4.80 UBD - SUBTRACT DOUBLE OPERANDS

SUBD (SRC1A, SRC2A, DSTA)

Parameters:

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

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

DSTA - Address of the quadword where the difference is stored

Result:

NOVALUE

## 4.81 SUBF - SUBTRACT FLOATING OPERANDS

SUBF (SRC1A, SRC2A, DSTA)

Parameters:

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

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

DSTA - Address of a longword where the difference is stored

Result:

NOVALUE

## 4.82 SUBG - SUBTRACT FLOAT-G OPERANDS

SUBG (SRC1A, SRC2A, DSTA)

Parameters:

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

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

DSTA - Address of a quadword where the difference is stored

Result

NOVALUE

## 4.83 SUBH - SUBTRACT FLOAT-H OPERANDS

SUBH (SRC1A, SRC2A, DSTA)

Parameters:

SRC1A - Address of an extended-exponent double-precision floating-point ocatword used as the subtrahend

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

DSTA - Address of an octaword where the difference is stored

Result:

NOVALUE

## 4.84 SUBM - SUBTRACT MULTIWORD OPERANDS

SUBM (SIZE, SRC1A, SRC2A, DSTA)

Parameters:

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

SRC1A - Address of an extended multi-precision integer used as the subtrahend

SRC2A - Address of an extended multi-precision integer used as the minuend

DSTA - Address of the difference

Result:

NOVALUE

## 4.85 TESTBITX - TEST AND MODIFY OPERATIONS

| | |
|---|---|
| TESTBITSS (FIELD) | Test for bit set, then set bit |
| TESTBITSC (FIELD) | Test for bit set, then clear bit |
| TESTBITCS (FIELD) | Test for bit clear, then set bit |
| TESTBITCC (FIELD) | Test for bit clear, then clear bit |
| TESTBITSSI (FIELD) | Test for bit set, then set bit (interlocked) |
| TESTBITCCI (FIELD) | Test for bit clear, then clear bit (interlocked) |

Parameter:

FIELD - Address with optional field selector which specifies a field whose low bit will be tested for a particular state. That bit will be set to the specified state. The size parameter of the field selector is ignored and a literal value one is substituted

Note that these are the only routines in the machine-specific set which accept an address with field selector that cannot be evaluated to an address.

Result:

1 - Bit tested was in the specified state

0 - Otherwise

## 4.86 XFC - EXTENDED FUNCTION CALL

XFC (OPCODE)

Parameters:

OPCODE - Link-time-constant-expression to be deposited in the byte which follows the XFC opcode in the instruction stream

Result:

NOVALUE

# CHAPTER 5

# PROGRAMMING CONSIDERATIONS

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

## 5.1 LIBRARY AND REQUIRE USAGE DIFFERENCES

BLISS library files are used like required files: declarations that are common to more than one module are centralized in a single file, which is automatically incorporated into other modules during compilation by means of REQUIRE or LIBRARY declarations.

Library files are more efficient for doing this than required files for two reasons. First, with files invoked by REQUIRE declarations, the cost of processing the source occurs every time the file is used in a compilation. However, with library files, the major compilation cost occurs once when the library is compiled, and a much smaller cost occurs each time the library file is used in a compilation. A library file closely approximates the internal symbol table representation used by the compiler; hence, costs of lexical processing (including scanning, lexical conditionals, lexical functions, and macro expansions) and declaration parsing and checking occur only during the library compilation.

Second, with files invoked by REQUIRE declarations, all declarations contained in the file are incorporated into the compiler symbol table. With library files, the compiler does not incorporate declarations into the normal symbol table until they are actually needed. Declarations of names that are not used do not fill up the symbol table.

The difference in cost depends on many factors, including the size of the library, the size of the module being compiled, and the percentage and kind of declarations used from a library. Experimental results indicate that compiler time and space requirements can typically be improved by a factor of four 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 compilation.

- Files invoked by REQUIRE declarations can contain any source text that is valid when that source text is substituted for the REQUIRE declaration. Files invoked by LIBRARY declarations must be compiled from sources that consist of a sequence of (only) the following declarations:

```
COMPILETIME
EXTERNAL
EXTERNAL LITERAL
EXTERNAL ROUTINE
FIELD
KEYWORDMACRO
LIBRARY
LINKAGE
LITERAL
MACRO
REQUIRE
STRUCTURE
SWITCHES
UNDECLARE
```

- SWITCHES declarations contained in files invoked by the REQUIRE declaration can affect the module being compiled; those contained in files used to produce library files affect only the library compilation. Switch settings are not incorporated into the compilation that uses the library file.

  (The only switches that are useful in a library compilation are LIST, LANGUAGE, and ERRS. The remaining switches may be given, but are effectively ignored since none of them has any effect on the declarations that are valid in a library compilation.)

- Files invoked by REQUIRE declarations can have effects that depend on previous declarations or switch settings in the module being compiled. This can occur in a lexical conditional (%IF-%THEN-%ELSE-%FI) or macro expansion that depends either on the lexical functions %SWITCHES, %DECLARED, or %VARIANT, or on values of predeclared literals, such as %BPVAL. (Refer to "Predeclared Literals" in Section 6.3.1.1.) Files invoked by LIBRARY declarations do not have effects that depend on previous declarations or switch settings, because SWITCHES declarations, REQUIRE declarations, lexical conditionals or macro calls contained in sources used to produce a library file are processed during the library compilation.

- Appropriately written source files can be invoked by REQUIRE declarations in BLISS compilers other than BLISS-32. Library files can be invoked only in compilations by the same compiler that created the library file.

In most normal cases, the source files used to create a library can be invoked by a REQUIRE declaration or the library can be invoked by a LIBRARY declaration with identical effects in the module being compiled. However, the differences presented above can lead to problems that are difficult to identify. Therefore, one or the other form should be used consistently for each set of declarations.

## 5.2 FREQUENT BLISS CODING ERRORS

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

### 5.2.1 Missing Dots

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

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

should almost certainly be:

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

### 5.2.2 Valued and Nonvalued Routines

The BLISS compiler does not diagnose useless value expressions in contexts that do not use a value. For example, 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;
```

an informational message is issued indicating that a value expression is missing in a context that implies a value. (The compiler assumes a value of zero for missing expressions.)

### 5.2.3 Semicolons and Values of Blocks

It is common to think of a semicolon as a terminator of an expression, but this is not true 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 5.2.2 above concerning valued and nonvalued routines.)

### 5.2.4 Complex Expressions Using AND, OR, and NOT

When writing complex tests involving the AND, OR, and NOT operators, it is easy to confuse the relative precedence of the operators. Use parentheses to make your intent explicit to the compiler, to other readers, and to yourself. For example, instead of:

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

use:

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

### 5.2.5 Computed Routine Calls

When computing the address of a routine to be called, enclose the expression that computes the address in parentheses followed by the parameters. For example:

```
BEGIN
EXTERNAL ROUTINE
    R;
LOCAL
    L;
L = R;
(.L)(0)
END
```

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:

```
BLISS(.L,0)
```

### 5.2.6 Signed and Unsigned Fields

Be careful when using signed and unsigned fields that are smaller than a fullword. Consistent use of the sign extension rules and signed versus unsigned operations is important. For example, 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.

### 5.2.7 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.

### 5.2.8 Missing Code

Many coding errors tend to result in code that can be optimized. If you discover that some of your program seems to be missing from the compiled code, you may possibly have made a coding error. Check the compiled code carefully to ensure that the code is indeed missing, rather than cleverly optimized.

For example, consider the following:

```
BEGIN
OWN
   X: BYTE;
...
IF .X EQL -5 THEN X = 0;
...
END
```

In the above example, the value of the test expression, .index EQL -5, is always false. (Refer to "Signed and Unsigned Fields" in Section 5.2.6.) As a result, the compiler

- produces no code for the test expression, and
- produces no code for the alternative, X=0 (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 SIGNED attribute in the declaration of X.

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

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

will always assign the value "1" to Y, because the low bit of ".index and 2" must always be zero (false).

## 5.3 ERRORS FROM LINKER

The TRUNC or TRUNCDAT error message is typically caused by a module compiled with the default addressing mode

```
ADDRESSING_MODE(WORD_RELATIVE)
```

being linked in an image that is larger than 32K. The problem can be resolved by any of the following procedures:

- Edit the offending source module to use ADDRESSING_MODE(LONG_RELATIVE) and recompile.
- Rearrange the placement of the object modules in such a manner that EXTERNAL or inter-PSECT references are within 32k bytes of their target.

## 5.4 OBSCURE ERROR MESSAGES

Obscure error messages that appear after a program has been run are typically caused by a program whose main routine fails to return a valid VMS completion code.

## 5.5 PIC CODE GENERATION

The BLISS-32 compiler always generates position-independent code (PIC) for expressions that involve relocatable quantities. This may cause surprisingly complex code to be generated. However, it is the programmer's responsibility to ensure that he/she has generated PIC data, when necessary.

# CHAPTER 6

# 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 "General Strategies" discusses some high-level approaches to writing transportable software in BLISS. The section "Tools" describes various features of the BLISS language that can be used in solving transportability problems. The section "Techniques for Writing Transportable Programs" 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 V1
BLISS-32 V2
BLISS-36 V2

BLISS-36 and BLISS-16 are intended as generic names for language that have BLISS compilers generating code for DEC-10/20's and PDP-11's, respectively.

## 6.1 INTRODUCTION

A transportable BLISS program is one that can be compiled to execute on at least two, and preferably all, of the three major architectures: PDP-10, PDP-11, and VAX-11. Various solutions to the problem of transportability exist, each requiring different levels of effort. Various kinds of solutions are recommended. For example, in some cases, program text should be rewritten. However, large portions of programs can be written in such a way that they will require no modification and yet be functionally equivalent in differing architectures. The levels of solutions in order of decreasing desirability, are:

- 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, it may be 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.

## 6.2 GENERAL STRATEGIES

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

- Isolation, and

- Simplicity

### 6.2.1 Isolation

Remember the following rule when you are designing or coding a program that is to be transported:

- If a construct is NOT transportable, isolate it.

You will probably encounter situations for which it is desirable to use machine-specific constructs in your BLISS program. In 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, the effort involved in transporting the program will be confined mainly to these easily identifiable, machine-specific sections. Specifically, follow these rules:

- If machine-specific data are to be allocated, place the allocation in a separate module or in a REQUIRE file.
- If machine-specific data are to be accessed, place the accessing code in a routine or 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 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 that are machine- or operating-system dependent from the rest of the system, the task of transporting the entire system is simplified. It becomes a matter of 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. The BLISS language 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.

### 6.2.2 Simplicity

A basic concept in writing transportable BLISS software is simplicity in the use of the language. BLISS was originally developed for the implementation of systems software. As a result of this, BLISS is one of few high-level programming languages that allow ready access to the machine on which the program will be running. 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 more complex. Reducing the complexity of data allocation, for example, results in a transportable subset of the BLISS language. This reduction of complexity is one of the basic themes that runs through these guidelines. In effect, the coding of 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. The use of the defaults in BLISS will result in programs that are much more easily transported.

## 6.3 TOOLS

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

### 6.3.1 Literals

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

**6.3.1.1 Predeclared Literals -** One of the key techniques in writing transportable programs is parameterization. Literals are a primary parameterization tool. The BLISS language has a set of predeclared, machine-specific literals that can be useful. These literals parameterize certain architectural values of the three machines. The values of the literals are dependent on the machine for which the program is currently being compiled. Here are their names and values:

| 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).

**6.3.1.2 User-Defined Literals** - Strictly speaking, a literal is not 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) | $\leq$ i $\leq$ (2**31) - 1 |
| 10/20: | -(2**35) | $\leq$ i $\leq$ (2**35) - 1 |
| 11: | -(2**15) | $\leq$ i $\leq$ (2**15) - 1 |
| ALL: | -(2**(%BPVAL-1)) | $\leq$ i $\leq$ (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 "Data: 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 an %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 "Data: Character Sequences."

### 6.3.2 Macros and Conditional Compilation

BLISS macros can be an essential tool in the development of transportable programs. Because they evaluate (expand) during compilation, it is possible to use macros to tailor a program to a specific machine.

A good example can be found in the section "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 select for compilation certain declarations 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
```

### 6.3.3 Module Switches

A module switch and a corresponding special switch are provided to aid in the writing of transportable programs. This switch, LANGUAGE, is provided for two reasons:

- To indicate the intended transportability goals of a module 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.
!-
```

### 6.3.4 Reserved Names

The following is a list of the BLISS reserved names. These names cannot be declared by the user. 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
```

### 6.3.5 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 and data declarations) in a transportable manner. Developing parallel REQUIRE files, one for each machine, can often provide a solution to transporting these constructs.

For example, if a certain set of routines 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 the use of the /LIBRARY switch, is capable of precompiling files containing declarations. However, not all declarations can be processed in a library run; those that are allowed are described 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 that is to run on a DECSYSTEM-20 and a VAX. To precompile it requires that it be run on the BLISS-32 compiler 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).

### 6.3.6 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, and so on, but the routine can still be easily used as a transportability abstraction mechanism.

For instance, when designing a system of transportable modules which uses the concept of floating-point numbers and associated operations, 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 significant, 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.

## 6.4 TECHNIQUES FOR WRITING TRANSPORTABLE PROGRAMS

This section on techniques shows you how to write transportable programs. The section is organized in dictionary form by BLISS construct or concept. Each subsection contains:

- A discussion of the construct or concept.
- Transportability problems that its use may engender.
- Specific guidelines and restrictions on the use of the construct or concept.
- Examples - both transportable and nontransportable.

In all cases, the examples attempt to use the tools described in the previous section.

### 6.4.1 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, and so on.) A presentation of more complex forms of data can be found in the sections "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.

**6.4.1.1 Problem Origin** - The allocation of data (via the OWN, LOCAL, GLOBAL, and other declarations) tends to be one of the most sensitive areas of a BLISS program in terms of transportability. This problem of transporting data arises chiefly from two sources:

- The machine architectures and
- 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 trying to write efficient machine-specific software. 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?

**6.4.1.2 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 his or her 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. In fact, 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 frequent use 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 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.

In the case of scalar data, 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 can not 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 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.

### 6.4.2 Data: Addresses and Address Calculations

This section will discuss address values and calculations using address values. First, there will be a presentation of the problems that might occur when using an address or the result of an address calculation as a value. A transportable solution to some of these problems is then presented. Next, a discussion of the need for address forms of the BLISS relational operators and control expressions and how and when to use them will be presented. Finally, some important differences in the interpretation of address values between BLISS-10 and BLISS-36 are discussed.

**6.4.2.1 Addresses and Address Calculations** - The value of an undotted variable name in BLISS is an address. In most cases, this address value is used only for the simple fetching and storing of data. When address values are used for other purposes, we must be concerned with the portability of an address or an address calculation. The term "address calculation" means any arithmetic operations performed on address values.

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 "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 the 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 "Structures and Field Selectors."

**6.4.2.2 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 (for example, testing for equality) 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, and so on) 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 of these guidelines can have unpredictable results.

**6.4.2.3 BLISS-10 Addresses Versus BLISS-36 Addresses** - There is a fundamental conceptual change from BLISS-10 to BLISS-36 in the defined value of a name. BLISS-10 defines the value of a data segment name to be a byte pointer consisting of the address value in the low half of a word, and position and size values of 0 and 36 in the high half of the word. BLISS-36, however, defines the value as simply the address in the low half and zeros in the high half. This change was made solely for reasons of transportability, 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 more thorough discussion can be found in the section "Structures and Field Selectors."

### 6.4.3 Data: Character Sequences

This section will discuss the use of character sequences (strings) in BLISS programs. Historically, there has been no consistent method for transportably dealing with strings and the functions operating upon them. Ad hoc string functions have been the rule, having been implemented by individuals or projects to suit their particular needs. This section will begin by looking at quoted strings in two different contexts. Transportability problems associated with quoted string, and guidelines for their use will be discussed.

Quoted strings are used in two different contexts:

- as values (integers) and
- as character strings

**6.4.3.1 Quoted Strings Used 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.

**6.4.3.2 Quoted Strings Used as Character Strings** - The necessity of using more than one character in a literal leads to the other situation in which quoted strings are used: as character strings.

To facilitate the allocation, comparison and manipulation of character sequences, a built-in character handling package has been constructed as part of the BLISS language. It has been implemented in BLISS-32, BLISS-36, and BLISS-16.

These built-in functions provide a 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.

### 6.4.4 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.

**6.4.4.1 PLITs in General** - Because BLISS values are a maximum of a machine word in length, any literal that requires more than a word for its value needs a separate mechanism, and that mechanism is the PLIT (or UPLIT). Hence, PLITs are a means for defining references to constants longer than one word. 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 these last three, providing that the value of each is known prior to execution time.

**6.4.4.2 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, and so on) 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 "Data." Further guidelines are presented in the section "Initializing Packed Data."

**6.4.4.3 String Literal PLIT Items** - The next guideline is based on the representation of PLITs in memory. Specifically the problem is encountered when scalar and string PLIT items appear in the same PLIT.

The difficulty arises primarily from the representation of characters on the different machines. A more thorough discussion of character representation can be found in the section "Data: 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)
```

on the 11, it is:

```
CONABC:                    / B A /   (16)
                           / D C /   (16)
                           /   E /   (16)
                         / address   / (16)
```

and 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 "Data: 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).

**6.4.4.4 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.

```
                    ...
                    ...
                    ...

    !+
    !       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 won't 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')

                );
        ...
        ...
        ...
```

**6.4.4.5 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 a field within a block is accessed, it is a common practice to associate each field reference (that is, offset, position and size) with a field name. So, for example, the field names for the original employee table would look like:

```
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 halfwords.

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 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.

The next page contains the example describing a method in which it could be initialized. Making it a BLOCKVECTOR has extended the structure. 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.

```
!+
!  DCB size parameters
!-

LITERAL
        DCB_NO_BLOCKS = total number of blocks,
        DCB_SIZE = size of a block;

!+
!  DCB Field Reference macros
!-

FIELD DCB =
    SET
    DCB_A = [0,0,8,0],
    DCB_B = [0,8,3,0],
    DCB_C = [0,11,5,0],
    DCB_D = [0,%BPVAL/2,%BPVAL/2,0],
    DCB_E = [1,0,%BPVAL,0];
    TES;
MACRO

!+
! Macro to create the shift value from the
! position parameter of a field reference macro
!-

        SHIFT(X) = %FIELDEXPAND(X, 2) %,

!+
! DCB initializing macro.
!  Five values are required.
!-
```

```
        DCB_INITIALIZE(A,B,C,D,E)[] =
                A^SHIFT(DCB_A) OR
                B^SHIFT(DCB_B) OR
                C^SHIFT(DCB_C) OR
                D^SHIFT(DCB_D) ,        ! first word

                E^SHIFT(DCB_E) %;       ! second word

!+
!  DCB Blockvector definition and initialization
!-

OWN
    DCB_AREA:
        BLOCKVECTOR[DCB_NO_BLOCKS, DCB_SIZE]
        INITIAL(
            DCB_INITIALIZE (
            1,2,3,4,                    ! first word
            5,                          ! second word

            6,7,8,9,                    ! first word
            10,                         ! second word
            ...
```

### 6.4.5 Structures and Field Selectors

Two BLISS constructs 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.

**6.4.5.1 Structures** - Structure declarations are sensitive to transportability in that one can specify parameters corresponding to characteristics of particular architectures. Also, in BLISS-32, the reserved words BYTE, WORD, LONG, SIGNED, and UNSIGNED have values of 1, 2, 4, 1 and 0 respectively when used as structure actual parameters.

The ability to specify architecture-dependent information can be an advantage in developing transportable structure declarations. Later in this section, a structure will be developed 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).

**6.4.5.2 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, the vector element size will be specified in terms of the number of bits. The existing VECTOR mechanism will not do this.

An example of its use would be to create a vector of 9-bit elements. The first decision that has to be made in its design is whether or not each element is to be exactly nine bits, or at least nine bits. For this example, we chose the smallest natural unit whose size is greater than or equal to nine 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 parameter (I) for accessing the vector, and an indication of whether or not the leftmost bit of an element is to be interpreted as a sign bit (EXT).

The access and allocation formal list for FLEX_VECTOR is:

```
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 allowed to assume only values of integer multiples of %BPUNIT (that is, 1*%BPUNIT, 2*%BPUNIT, and so on), only a structure-size expression of the following form would be needed:

```
[ N * (UNIT) / %BPUNIT ]
```

Dividing the element size (UNIT) by %BPUNIT would give the size of each element in the vector in terms of an integer multiple. This value would then be multiplied by the number of elements to give the total size of the data to be allocated.

Suppose the structure needs to be more flexible in that it should be possible to specify any size element (within certain limits). The structure size must be slightly more complex:

```
[ N * ((UNIT + %BPUNIT - 1)) / %BPUNIT ]
```

The structure-size expression now computes enough %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
```

that 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 consists of the name of the structure (the base address) plus an offset based on the index I. In addition, a field selector is needed to access the proper number of bits at the calculated address.

The offset is simply the expression NO_OF_UNITS multiplied by 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<br>FLEX_VECTOR<0,0,1> |
| UNIT = 1 to 8 | [N * 1] Bytes<br>(FLEX_VECTOR + I)<0,8,1> |
| UNIT = 9 to 16 | [N * 2] Bytes<br>(FLEX_VECTOR + I * 2)<0,16,1> |
| UNIT = 17 to 32 | [N * 4] Bytes<br>(FLEX_VECTOR + I * 4)<0,32,1> |

| | |
|---|---|
| 11 | |
| UNIT = 0 to 16 | same as VAX-11 |
| 10/20 | |
| UNIT = 0 | no storage<br>(FLEX_VECTOR)<0,0,1> |
| UNIT = 1 to 36 | [ N ] Words<br>(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 that has this particular structure attribute are accessed very efficiently because they are always on addressable boundaries. Also, they are always some multiple of an addressable unit in length.

If this structure were to access elements of exactly the size specified, 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.

6.4.5.3 **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 |
|---|---|---|---|
| | $0 \leq p$<br>$p + s \leq 36$<br>$0 \leq s \leq 36$ | $0 \leq p$<br>$p + s \leq 16$<br>$0 \leq s \leq 16$ | $0 \leq s \leq 32$ |

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 can appear only in the definition of user-defined structures.

Restricting the domain of field selectors to structures isolates their use. Field selectors should be isolated so that:

- Changes in data structure design are easier.
- Machine dependencies can easily be placed in REQUIRE files.
- Complex coding making heavy use of the predeclared literals is limited to declarations.

Another transportable structure will be developed and will be affected by the table of field selector value restrictions.

**6.4.5.4 GEN_VECTOR** - Notice that FLEX_VECTOR does not attempt to pack data. Using the example of 9-bit elements, it is evident that there will be some wasting of bits - from seven bits on the 11 and VAX-11 to 27 on the 10/20 systems.

A variation of FLEX_VECTOR can be developed to provide a certain degree of packing. For example, in the case of 9-bit elements it would be possible to pack at least four of them into a 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. However, 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 are 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 runtime, 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.

**6.4.5.5 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 and
- 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 be used only in structure declarations.

# CHAPTER 7

# 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.

## 7.1 COMPILER PHASES

The compiler is organized conceptually into seven major phases:

| | |
|---|---|
| LEXSYN | - Lexical and syntactic analysis |
| FLOW | - Flow analysis |
| DELAY | - Heuristics |
| TNBIND | - Temporary name binding (register allocation) |
| CODE | - Code generation |
| FINAL | - Code stream optimization |
| OUTPUT | - Object and listing file production |

This division of the compiler into conceptual phases corresponds only approximately to the actual compiler. In some cases, a phase actually consists of two or more subphases. In other cases, phases are combined in the implementation. This level of detail is not important in the following discussion of the phases. The term "phase" should not be taken literally.

### 7.1.1 Lexical and Syntactic Analysis

The lexical and syntactic phase, LEXSYN, performs the following functions:

- Reads the input files
- Divides the source character text into lexemes
- Identifies and performs lexical functions and macro expansions
- Parses the resulting input lexeme sequence and creates appropriate symbol table entries for declarations and tree representations for expressions

The BLISS compiler reads the source text once and uses it to create an internal representation of the module. In this sense, the BLISS compiler is a one-pass compiler. On the other hand, at the end of each (ordinary or global) routine definition, the remaining phases of the compiler are performed in turn to analyze and completely produce and output code for that entire routine. In this sense, the BLISS compiler is a multi-pass compiler.

If the NOCODE switch is specified, the compiler operates in a "syntax-only" mode, in which the LEXSYN phase does not produce the tree representations for expressions and the later phases are not performed. If an error (as 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. However, there are two important limitations to such use:

1. Some errors are detected and reported by later phases of the compiler. Since the later phases are not performed in this mode, some errors will not be detected.

2. Because the tree representation of expressions is not being produced, the values of compile-time constant expressions cannot be determined. The compiler assumes a value of 0 for any expression that is not a simple literal. If the correct parsing of the module depends on the correct values of constant expressions, spurious error diagnostics can result. Examples where this might be the case are lexical functions, conditional compilation, and macro expansion.

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.

### 7.1.2 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 can be evaluated once before starting the loop and the result used during each iteration, thereby saving execution time.

- Identifies expressions that occur on all alternatives of the IF, CASE, and SELECTONE expressions. Such expressions 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.

**7.1.2.1 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 always the case that such indirect assignments are used to change the contents of

- dynamically created data structures that do not have names, or
- 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 value of the /OPTIMIZE qualifier can 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.

**7.1.2.2 Accounting for Changes** - The BLISS language definition intentionally leaves unspecified the order of operand evaluation in operator expressions in order to permit maximum optimization by the BLISS compiler. For example, the expression

```
F(X) + .X
```

can be evaluated first, by calling F with the address X as a parameter, second, by fetching the contents of X, and finally, by performing the addition. It might also be evaluated first, by fetching the contents of X, second, by calling F, and so on. The compiler uses information about the entire routine in which the expression is contained to choose alternatives. 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.

```
          !
BEGIN exp ;... END

      !       !       !
IF exp THEN exp ELSE exp

         !     !
WHILE exp DO exp

      !        !
DO exp WHILE exp

                  !     !     !     !
INCR name FROM exp TO exp BY exp DO exp

        !                                   !
CASE exp FROM ctce TO ctce OF SET [ ... ]: exp ;... TES

          !           !     !          !
SELECT exp OF SET [ exp TO exp ,... ]: exp ;... TES
```

The most common mark point in most programs is the semicolon, which separates expressions in a block or compound expression, for example:

```
BEGIN
Y = .X+2;
Z = .X+2+F(X);
W = .X+2
END
```

In the second line, the content of Y is changed. 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 mark points 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.

### 7.1.3 Heuristic Phase

The heuristic phase, DELAY, further analyzes the routine to obtain general information about the routine. Some of this information is used by DELAY itself to make optimization decisions, and some is made available for use by later phases. DELAY performs the following functions:

- Evaluates the effectiveness of the alternatives identified by FLOW and chooses the best alternative. This analysis considers, for example, the number of occurrences of a common subexpression and the potential for using specialized operations available in the address parts of instructions (for example, indirection and indexing).

- Identifies sets of 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.

### 7.1.4 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 functions:

- 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 can become larger.

### 7.1.5 Code Generation

The code generator 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.

None of the compilation switches affect the operation of the CODE phase.

### 7.1.6 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.
- Test elimination. TST instructions used for conditional branching may be deleted if the condition codes needed by the branch instruction are appropriately set by the instruction(s) that precede the TST.
- Jump/Branch resolution. Branch instructions are reduced to their shortest size consistent with the actual range of the needed branch displacement.
- Branch chaining. If a short branch instruction is not able to reach the desired location but can reach another branch instruction that goes to the desired location, a chain of branches is used to minimize code size.
- Cross-jumping. Identical sequences of code that flow into a common instruction are merged into a single sequence.

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.

### 7.1.7 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.

## 7.2 SUMMARY OF SWITCH EFFECTS

The previous sections have described the phases of the compiler and the switches that affect each of those phases. This section summarizes the effects of each switch throughout the compiler.

| Switch Name | Phase | Effect |
|---|---|---|
| CODE | LEXSYN | NOCODE specifies syntax-only processing; the other phases are not invoked. |
| OPTIMIZE | FLOW | If flow analysis is performed, NOOPTIMIZE specifies do not optimize across mark points. |

| Switch Name | Phase | Effect |
|---|---|---|
| OPTLEVEL | FLOW | At levels 0 and 1, flow analysis is not performed. |
| | FINAL | At levels 0 and 1, cross-jumping and branch chaining are not performed. At level 0, peephole optimizations that are not adjacent are not performed. |
| SAFE | FLOW | If flow analysis is performed, SAFE specifies that indirect changes are to assume all storage is changed. |
| 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, and so on) to override OPTLEVEL at any time. In a compilation of more than one module, each module begins with the setting defined in the command line and OPTLEVEL=2 if OPTLEVEL has not been specified in the command line.

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, and so on | | | X | X |
| 3. Targetting/preferencing to temporaries | X | X | X | + |
| 4. Cross jumping | | | X | X |
| 5. Multiple RETs (instead of only one return point) | | X | X | X |
| 6. Peepholes | - | X | X | X |
| 7. TSTx elimination | X | X | X | X |
| 8. Scans for ()+, -() forms | X | X | X | X |
| 9. Scans for PUSHR, and so on | X | X | X | X |
| 10. Scans for cheaper form of addressing | X | X | X | X |
| 11. Branch chaining | | | X | X |
| *12. "SAFE" optimizations | | | X | X |
| *13. "OPTIMIZE" over mark points (for example, ;) | | | X | X |
| *14. "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 8

# TOOLS, LIBRARIES, AND SYSTEM INTERFACES

A number of programming tools, libraries, and system interfaces are available for use by BLISS programmers. This chapter briefly describes what is available for use with BLISS-32 V3. Note that the BLISS tools (utility programs) and system interfaces described here **are not supported products** at the time of writing.

The residence locations specified for the various BLISS-related program and information files assume that these files have been installed on your system in their standard, as-released locations.

## 8.1 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.

### 8.1.1 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. |

### 8.1.2 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. |

### 8.1.3 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 |

### 8.1.4 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. |

### 8.1.5 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.

## 8.2 BLISS CROSS REFERENCES (BLSCRF)

The utility program BLSCRF cross-references BLISS source files and provides a means of merging the cross-references of multiple files.

### 8.2.1 Command Line Format

BLSCRF is invoked by defining the string-symbol:

```
BLSCRF:==$BLSCRF
BLSCRF in-spec{/OUTPUT:outspec}{/MERGE:merge-spec}{qualifier...}
```

Wildcards are permitted in the directory, file name, file type, and file version fields. A comma-list is not accepted by the current implementation.

The default file types are:

```
in-spec     B32 (BLI)
out-spec    XRF
merge-spec  MRG
```

### 8.2.2 Command Semantics

BLSCRF accepts as input BLISS source text and produces a cross-reference listing file (out-spec) and/or merge file (merge-spec) suitable for combining with similar files to produce a master cross-reference of multiple modules.

The cross-reference listing is a numbered listing of the input text, followed by an alphabetical listing of symbols and line numbers on which the symbols occur. This file is suitable for listing on a 132-column line printer.

The merge file contains cross-references by symbol and file name. This can be appended to other merge files and sorted to create a master cross-reference.

### 8.2.3 Command Qualifiers

A number of command-line qualifiers can be specified to permit some flexibility in what is cross-referenced and how it is formatted (* = default).

/FLAG:(ASSIGN,REQUIRE,ROUTINE)

Certain characters can be used to flag various symbol uses, including:

| | |
|---|---|
| ASSIGN*<br>NOASSIGN | "#" indicates that symbol appeared on left side of an equals sign and represents a probable assignment |
| REQUIRE*<br>NOREQUIRE | "+" indicates that symbol appeared in REQUIRE file |
| ROUTINE*<br>NOROUTINE | "*" indicates that symbol appeared as a routine-name in a ROUTINE declaration (also FORWARD ROUTINE, BIND ROUTINE, or EXTERNAL ROUTINE) |

| | |
|---|---|
| /KEYWORD<br>/NOKEYWORDS* | The use of BLISS language keywords is cross referenced. |
| /MERGE:file-spec | Create a merge file. |
| /ONCE<br>/NOONCE* | Cross reference only those symbols that appear exactly once in the input file. |
| /OUTPUT:file-spec | Specifies output file. |
| /REQUIRE<br>/NOREQUIRE* | List source lines in REQUIRE files. |
| /SOURCE*<br>/NOSOURCE | Include a line numbered listing of the source with the cross-reference in the output file. |

## 8.3 BLISS LANGUAGE FORMATTER (PRETTY)

PRETTY is a utility program that reformats BLISS source files and require files so that they correspond to the formatting standards described in the VAX-11 Software Engineering Manual. PRETTY is a valuable tool in standardizing the appearance of BLISS code, for it promotes readability while permitting flexibility in program structure.

PRETTY is more than a reformatter. It also performs a cursory syntax check of your program and reports obvious errors that should be fixed prior to compilation, such as mismatched BEGIN and END statements.

### 8.3.1 Command Line Format

PRETTY is invoked by defining the string symbol:

```
PRETTY:==$PRETTY
PRETTY in-spec{/OUTPUT: out-spec}{/LISTING: list-spec},...
```

### 8.3.2 Command Semantics

The output file contains the reformatted source program or REQUIRE file. Some of the operations performed by PRETTY are:

- To force routines to the top of a new page (if they are not nested within other routines)
- To align BEGIN-END pairs
- To indent blocks
- Where possible, to align end-of-line comments (remarks) to a standard column
- To align block comments with the enclosing blocks
- To realign control structures to one or more lines in a standard way

Several options are available to enable a user to specify different kinds of formatting; defaults for all options are described.

The listing file differs from the output file in three ways:

- Each line is appended with a slash (/), the SOS page number, and the SOS line number, as in:

```
/  4.  200
```

- Each logical tab is preceded by a colon (:), so that the indentation level is obvious, as in:

```
IF .A                          /  1.  1200
THEN                           /  1.  1300
:                              /  1.  1400
:   IF .B                      /  1.  1500
:   THEN                       /  1.  1600
```

```
:   :                                   /  1.   1700
:   :   IF .C                           /  1.   1800
:   :   THEN                            /  1.   1900
:   :   :                               /  1.   2000
:   :   :   D=5;                        /  1.   2100
```

- There is a three line heading on each page:
  - %TITLE information (which starts a new page)
  - %SBTTL information
  - Module and routine names and page number

The listing file cannot be used as input to the BLISS compiler.

On VAX, if no output or listing files are specified, only terminal error messages are produced. Otherwise, an output file is created, if specified, and a listing file is created, if specified. If no input file extension is specified the extension defaults to .BLI.

Producing both an output file and listing file doubles the execution time of PRETTY. However, PRETTY's typical execution time is only a fraction of the BLISS compiler's.

### 8.3.3 Formatting Options

You can specify in the source input a number of options, which give you flexibility in deciding how to format your code. You supply formatting options by inserting directives as full-line comments in the input source text. The format of the formatting option directive is:

```
!<BLF/option>
```

The exclamation point that starts the command must begin in column 1. Options may be typed in uppercase or lowercase characters. Commands are passed to PRETTY in this fashion without modification to either the output or the listing files. Since the directives begin with an exclamation point, they are interpreted as comments by the BLISS compiler.

Formatting option directives are described below. An asterisk (*) identifies the default.

ERROR*

This enables the insertion of error messages (that is, lines beginning with "!!ERROR!!") into the output file. Error messages are automatically deleted from source files on subsequent runs.

Example: !<BLF/ERROR>

NOERROR

This disables the insertion of error messages into the output file. Error messages are always output to the terminal.

Example: !<BLF/NOERROR>

FORMAT*

This causes resumption of formatting by PRETTY.

Example: !<BLF/FORMAT>

NOFORMAT

This inhibits formatting by PRETTY until the next !<BLF/PAGE> or !<BLF/FORMAT> is encountered.

Example: !<BLF/NOFORMAT>

LOWERCASE

This causes all identifiers and keywords to be converted to lowercase.

Example: !<BLF/LOWERCASE>

LOWERCASE_KEY

This causes BLISS keywords, builtins, and predefined names to be converted to lowercase.

Example: !<BLF/LOWERCASE_KEY>

LOWERCASE_USER

This causes user identifiers to be converted to lowercase.

Example: !<BLF/LOWERCASE_USER>

MACRO

This causes macros to be formatted. Use of this option may cause error messages that do not appear when formatting with macros is disabled.

Example: !<BLF/MACRO>

NOMACRO*

This disables formatting of macros.

Example: !<BLF/NOMACRO>

NOCASE*

This causes all identifiers and keywords to remain unchanged.

Example: !<BLF/NOCASE>

NOCASE_KEY

This causes all BLISS keywords, builtins, and predefined names to remain unchanged.

Example: !<BLF/NOCASE_KEY>

NOCASE_USER

This causes all user identifiers to remain unchanged.

Example: !<BLF/NOCASE_USER>

PAGE

A page break (formfeed) is generated.

Example: !<BLF/PAGE>

PLIT

This causes formatting of PLIT bodies to occur. Since the content of PLITs is difficult to analyze, PRETTY may format the PLIT differently than you had intended.

Example: !<BLF/PLIT>

NOPLIT*

This inhibits formatting of PLIT bodies. Only the first line of a PLIT body is formatted; the remaining lines remain unchanged.

Example: !<BLF/NOPLIT>

REMARK:n (Default = 6)

This causes subsequent comments to begin at column 8*n+1, where 2 < n < 16. The default comment starts at column 49.

Example: !<BLF/REMARK:5>

REQUIRE'file-spec'

The specified file is read for further formatting option directives. Everything in the REQUIRE file other than legal directives is ignored. Directives in the REQUIRE file are not reproduced in the output or listing files. The REQUIRE file must not contain another REQUIRE directive.

Example: !<BLF/REQUIRE'COMMAND.REQ'>

SYNONYM name = lexeme...

This option describes to PRETTY macros that you have defined to substitute for keywords. SYNONYM causes subsequent occurrences of "name" to be treated as a sequence of other tokens for formatting purposes. The special token "*" can be used to indicate where to position "name" with regard to the normal position of the tokens that it replaces.

Example: !<BLF/SYNONYM ELIF = ELSE IF *>

permits the sequence:

```
IF expr THEN expr ELIF expr THEN expr;
```

to be formatted without error. Only the name ELIF is output, but it is indented as though ELSE IF had occurred in the text.

In the above example, formatting would occur as follows:

```
IF expr
THEN
    expr
    ELIF expr
    THEN
        expr;
```

If !<BLF/SYNONYM ELIF = ELSE * IF> had been specified, formatting would occur as:

```
IF expr
THEN
    expr
ELIF
    expr
    THEN
        expr;
```

UPPERCASE

This causes all identifiers and keywords to be converted to uppercase.

Example: !<BLF/UPPERCASE>

UPPERCASE_KEY

This causes all BLISS keywords, builtins, and predefined names to be converted to uppercase.

Example: !<BLF/UPPERCASE_KEY>

UPPERCASE_USER

This causes all user identifiers to be converted to uppercase.

Example: !<BLF/UPPERCASE_USER>

WIDTH:n (Default = 110)

This causes subsequent output lines to have a width of "n" columns, where 71 < n <141.

Example: !<BLF/WIDTH:80>

### 8.3.4 Hints on Using PRETTY

Because of the great flexibility possible in the use of the BLISS language, specification of fixed formatting rules for the language is difficult. In order to construct a formatting tool like PRETTY, an idealized model of a BLISS program must be used. This model may be quite different from your particular style of manual formatting.

Several features of the language can be used in such a way as to make formatting difficult without very extensive semantic analysis. In order to meet its goals of executing many times faster than the compiler, PRETTY does not perform this kind of extensive analysis, but rather relies on hints from the programmer.

**8.3.4.1 Breaking Lines** - PRETTY attempts to break lines by certain general rules. In a begin-block, every semicolon terminating a main line expression causes a line break except if a remark follows. Therefore, a remark can be used to force a line break.

Operator-expressions that are too long to fit on a single line will be broken around an operator. A programmer can specify an alternate line break strategy by using the exclamation point as a break character. Consider a (visually) long expression of the form:

```
E1 OR ... AND EN
```

where Ei denotes an expression.

If the programmer wishes to control how this line is broken, the following can be written:

```
E1    OR    ! comment
E2    OR    !
...   AND   !
EN          !
```

PRETTY attempts to place an if-expression on a single line if it fits. If it does not, PRETTY will automatically place the 'THEN' under the 'IF', and the 'ELSE' if present, under the 'THEN'.

**8.3.4.2 Comments** - While BLISS recognizes only two kinds of comments, embedded comments ( %( ... )% ) and trailing comments ( ! ... ), PRETTY distinguishes several subtypes of trailing comments:

- Remarks:

  ```
  ...! comment
  ```

  These are lined up in the remark column, but otherwise not analyzed. They cause a line break.

- Full line comment:

  ```
  ! rest of line ...
  ```

  This occurs in column 1. The entire line is passed to the output file without further processing.

- Offset BLOCK comments:

  ```
  ...

  !+
  ! comment
  !-

  ...
  ```

  These are preceded and followed by a blank line, and indented to the current indentation level.

- Non-offset BLOCK comments:

```
...
!
! comment
!
...
```

These are similar to offset block comments in that they are similarly indented, but are not offset from the surrounding text by blank lines. They are recognized as block comments rather than remarks only if they appear more than one tab to the left of the remark column.

8.3.4.3 **MACROs** - Because it is possible to write macros in BLISS that are not expressions or lists of expressions, PRETTY conservatively makes no attempt to format macro bodies, and treats macro invocations in the same way as routine calls. Thus, the macro body appears in the output file the same way it appeared in the input file, and thus, may be formatted quite differently from surrounding text in the output file.

Experience has shown, however, that the great majority of macro bodies can be successfully formatted by PRETTY. Because of this, you may want to change the default to:

```
!<BLF/MACRO>
```

for your programs by preceding each source file with the above command. Macro formatting can be turned off before any offending macro declaration, and turned on after it, should there be any.

The SYNONYM facility allows users a limited way of telling PRETTY to interpret certain macro invocations as other than routine calls, as in the ELIF macro given above.

8.3.4.4 **PLITs** - PLITs are used to construct initialized tables in BLISS, and in practice, the programmer-specified formatting of a table gives a good indication to the reader of the structure and meaning of the table.

Rather than try to guess the structure of a PLIT, PLIT formatting is turned off by default.

The initial-attribute of a declaration is handled exactly as a PLIT.

## 8.4 TUTORIAL TERMINAL INPUT/OUTPUT PACKAGE (TUTIO)

TUTIO.R32 is a BLISS REQUIRE file that contains some simple terminal I/O primitives. This package is normally used in 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 'SYS$LIBRARY: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.

The TUTIO package will function only with a terminal. It will not work from a batch job or command-procedure file.

## 8.5 VAX/VMS SYSTEM SERVICES INTERFACE

System services are procedures used by the operating system to control resources available to processes, to provide for communication among processes, and to perform basic operating system functions, such as the coordination of input/output operations. Although most system services are used primarily by the operating system itself on behalf of logged-on users, many are available for general use and provide techniques that can be used in application programs.

The BLISS interface to the VAX/VMS System Services routines is in SYS$LIBRARY:STARLET.REQ or SYS$LIBRARY:STARLET.L32. The primary function of the interface is to guarantee the proper number and sequence of arguments required for the System Service calls.

Using the BLISS interface macros, the syntax for invoking system services in BLISS is almost identical to the VAX-11 MACRO keyword syntax. The major difference (aside from the fact that BLISS expressions appear in the parameter list) is that BLISS requires the macro parameter list to be enclosed in parentheses. (See the example of the BLISS-style usage below.)

To use the interface, include the declaration:

```
LIBRARY 'SYS$LIBRARY:STARLET'
```

System services act like routine calls and return a condition-value as their result. The following code fragment checks for successful completion of the system service:

```
IF STATUS=$WAITFR(EFN=1)
THEN
    ...                 !successful wait
ELSE                    !invalid event flag
    SIGNAL(.STATUS);    !pass back failure
```

For detailed information on available services, see the VAX/VMS System Services Reference Manual.

### 8.5.1 Sample Program Using VMS System Services

The program shown below prints the time of day at the user's terminal. Although not very interesting as a practical program, it does show the use of two system services, $GETTIM and $FAO, and of the common run-time library routine LIB$PUT_OUTPUT.

```
MODULE SHOWTIME(IDENT='1-1' %TITLE'Print time', MAIN=TIMEOUT)=
BEGIN
LIBRARY 'SYS$LIBRARY:STARLET';        ! Defines System Services

OWN
    TIMEBUF: VECTOR[2],                    ! 64-bit system time
    MSGBUF: VECTOR[80,BYTE],               ! Output message buffer
    MSGDESC: BLOCK[8,BYTE] PRESET( [DSC$W_LENGTH]=80,
                                   [DSC$A_POINTER]=M36BUF)
BIND
    FMTDESC=%ASCID %STRING('At the tone, the time will be ',
            %CHAR(7), '!15%T' );
EXTERNAL ROUTINE
    LIB$PUT_OUTPUT: ADDRESSING_MODE(GENERAL);

ROUTINE TIMEOUT=
    BEGIN
    LOCAL
        RSLT: WORD;                        ! Resultant string length

    $GETTIM( TIMADR=TIMEBUF );             ! Get time as 64-bit integer

    $FAOL( CTRSTR=FMTDESC,                 ! Format Descriptor
        OUTLEN=RSLT,                       ! Resultant length (only a word!)
        OUTBUF=MSGDESC,                    ! Output buffer descriptor
        PRMLST= %REF(TIMEBUF));            ! Addr of 64-bit time block
    MSGDESC[0] = .RSLT;                    ! Modify output descriptor
    LIB$PUT_OUTPUT( MSGDESC )              ! Return status
    END;
END ELUDOM
```

In the SHOWTIME example, LIB$PUT_OUTPUT is part of the VMSRTL shareable-image, as such it is referenced via GENERAL addressing. The call on $FAOL uses %REF to make the PRMLST value the address of a pointer to the TIMEBUF.

### 8.5.2 Common Errors in Using System Services

When the user invokes system services, he can encounter a number of errors. Two of the more common errors occur when the $HIBER and $FAO services are invoked.

The $HIBER system service keyword macro has no arguments. Thus, it is invoked as:

```
$HIBER
```

Notice the absence of any parenthesized argument list.

The OUTLEN keyword parameter of the $FAO and $FAOL system service is the address of a WORD (16-bit) value. A common error is to provide that parameter with the address of a fullword (32-bit) value, rather than a WORD value. Thus, the 16 high-order bits always contain a variable, indeterminate value. Consequently, when the full 32-bit value is used rather than just the 16 low-order bits, unpredictable results occur.

Users should guard against this problem in calls to other system services.

## 8.6 RECORD MANAGEMENT SERVICES INTERFACE

VAX-11 Record Management Services (RMS-32) is a collection of file- and record-level I/O routines that enable user programs to process and manage data files. Also included as a part of RMS-32 is a set of BLISS macro definitions that facilitate control-block initialization and the calling of RMS control routines.

RMS definitions appear in SYS$LIBRARY:STARLET.REQ or SYS$LIBRARY:STARLET.L32. For a general description of RMS-32 programming, see the VAX-11 Record Management Services User's Guide and VAX-11 Record Management Services Reference Manual.

### 8.6.1 Using RMS-32 Macros

STARLET.REQ provides three types of RMS definitions:

a. Field definitions and constants used in RMS control blocks

b. Static and dynamic control-block initialization macros

c. RMS routine-call keyword macros.

The best way to understand these macros is to print a copy of STARLET.REQ and study it.

Macros for defining FAB, RAB, and XAB blocks are provided. Each control block has macros for static or dynamic initialization and for uninitialized structures; for example:

| | |
|---|---|
| $FAB | A keyword macro for static definition. |
| $FAB_INIT | A keyword macro for run-time dynamic initialization. |
| $FAB_DECL | A simple macro for FABs that require no initialization. Typically, this is used to map attributes onto a routine's formal parameter. |

Keywords for these macros are identical to those used by the VAX-11 MACRO RMS-32 interface, except for those keywords that expect a 64-bit value. These keywords are handled as "key0" and "key1", each of which has a 32-bit value.

### 8.6.2 Sample Routine Using RMS-32

The following sample program illustrates the use of the RMS macros. The program opens a file named MYFILE.SRC and copies it to a file named MYFILE.LIS.

```
MODULE RMSTEST(IDENT='1-1' %TITLE'BLISS-RMS Example',MAIN=COPYIT)=
BEGIN
!++
! Sample program showing use of BLISS/RMS-32 Interface Macros
!--
LIBRARY 'SYS$LIBRARY:STARLET';          ! All definitions are here

OWN
    MYBUF:    VECTOR[CH$ALLOCATION(132)],    ! Input record buffer

    INFAB:  $FAB(FNM='MYFILE.SRC',           ! Source FAB
                 FAC=GET),

    OUTFAB: $FAB(FNM='MYFILE.LIS', RFM=VAR, ! Destination FAB
                 RAT=CR, FAC=PUT),

    INRAB:  $RAB(UBF=MYBUF, USZ=132,         ! RAB def'n: Locate Read,
                 ROP=<LOC,RAH>, FAB=INFAB), ! and Multibuffer I/O
    OUTRAB: $RAB(ROP=WBH, FAB=OUTFAB);       ! on input and output.

ROUTINE COPYIT=
    BEGIN
    LOCAL
       STS;         ! RMS service-completion code

    $OPEN( FAB=INFAB );       ! Open input file
    $CONNECT( RAB=INRAB );    ! and associate RAB.
    $CREATE( FAB=OUTFAB );    ! Create a new output file
    $CONNECT( RAB=OUTRAB );   ! and associate a RAB.
    !+
    ! Copy loop. Read records from input until
    ! EOF or error encountered. Write each record as it is read.
    !-
    WHILE ( STS=$GET( RAB=INRAB ) ) DO
         BEGIN
         OUTRAB[RAB$L_RBF] = .INRAB[RAB$L_RBF];  ! Copy record descr.
         OUTRAB[RAB$W_RSZ] = .INRAB[RAB$W_RSZ];  ! to output RAB and
         $PUT( RAB=OUTRAB )                      ! write the record.
         END;
```

```
    $CLOSE( FAB=INFAB );        ! Close the input
    $CLOSE( FAB=OUTFAB );       ! and output files.

    IF .STS EQL RMS$_EOF        ! Check for EOF on input
    THEN
        SS$_NORMAL              ! Tell System everything is OK, or
    ELSE
        .STS                    ! Return the RMS error status.
    END;
END ELUDOM
```

## 8.7 OTHER VAX/VMS INTERFACES

### 8.7.1 LIB

LIB.L32 contains internal interfaces between BLISS-32 and the VAX/VMS Operating System as well as everything in STARLET.L32. Much of LIB is normally useful only to an operating-system programmer. Most users should use SYS$LIBRARY:STARLET.L32, the user relevant subset of LIB.

### 8.7.2 TPAMAC

TPAMAC.REQ provides BLISS macros for LIB$TPARSE, a finite-state transition parser. The sample program presented in Figure 8-1 closely parallels the example in Chapter 8 of the VAX-11 Runtime Library Reference Manual.

```
MODULE CREATE_DIR(%TITLE 'Create Directory File' IDENT = 'X0000',
        MAIN=CREATE_DIR) =
BEGIN

!+
!
! This is a sample program that accepts and parses the command line
! of the CREATE/DIRECTORY command.
! This program contains the operating system
! call to acquire the command line from the command interpreter
! and parse it with TPARSE, leaving the necessary information
! in its global data base. The command line has the following format:
!
!       CREATE/DIR DEVICE:[MARANTZ.ACCOUNT.OLD]
!               /OWNER_UIC=[2437,25]
!               /ENTRIES=100
!               /PROTECTION=(SYSTEM:R,OWNER:RWED,GROUP:R,WORLD:R)
!
! The three qualifiers are optional. Alternatively, the command
! may take the form
!
!       CREATE/DIR DEVICE:[202,31]
!
! using any of the optional qualifiers.
!
!-
```

**Figure 8-1: Sample TPARSE Program**

```
LIBRARY 'SYS$LIBRARY:STARLET';          ! Define CLI offsets
LIBRARY 'SYS$LIBRARY:CLIMAC';           !  and request blocks,
LIBRARY 'SYS$LIBRARY:TPAMAC';           !  and TPARSE stuff too.


FORWARD ROUTINE
        BLANKS_OFF,
        CHECK_UIC,
        STORE_NAME,
        MAKE_UIC,
        CREATE_DIR;

!
! Define parser flag bits for flags longword
!
LITERAL
    UIC_FLAG=           1,              ! /UIC seen
    ENTRIES_FLAG=       2,              ! /ENTRIES seen
    PROT_FLAG=          4;              ! /PROTECTION seen

!
! CLI request descriptor block to get the command line
!
OWN
    REQ_COMMAND: $CLIREQDESC( RQTYPE=GETCMD );

!
! TPARSE parameter block
!
OWN
    TPARSE_BLOCK: BLOCK[TPA$C_LENGTH0,BYTE]
                  PRESET( [TPA$L_COUNT]=TPA$K_COUNT0,    ! Longword count
                          [TPA$L_OPTIONS]= TPA$M_ABBREV ! Allow abbreviation
                                OR TPA$M_BLANKS);       ! Process spaces explicitly

!
! Parser Data Base
!
OWN
    PARSER_FLAGS,                                       ! Keyword flags
    DEVICE_STRING:      VECTOR[2],                      ! device string descriptor
    ENTRY_COUNT,                                        ! space to preallocate
    FILE_PROTECT,                                       ! directory file protection
    UIC_GROUP,                                          ! temp for UIC group
    UIC_MEMBER,                                         ! temp for UIC member
    UIC_OWNER,                                          ! actual file owner UIC
    NAME_COUNT,                                         ! number of directory names
    DIRNAME:    BLOCKVECTOR[8,2,LONG];                  ! name descriptors 0-7


EXTERNAL ROUTINE
    SYS$CLI:    ADDRESSING_MODE(GENERAL),
    LIB$TPARSE: ADDRESSING_MODE(GENERAL);
```

**Figure 8-1 (Cont.): Sample TPARSE Program**

```
%SBTTL 'Parser State Table'

$INIT_STATE( UFD_STATE, UFD_KEY );

!
! Read over the command line to the first blank in the command.
!
        $STATE( START,
          (TPA$_BLANK,          ,BLANKS_OFF),
          (TPA$_ANY,    START));

!
! Read device name string and trailing colon
!
        $STATE(,
          (TPA$_SYMBOL,         ,       ,       ,DEVICE_STRING));
        $STATE(,
          (':'));
!
! Read directory string, which is either a UIC string or a general
! directory string.
!
        $STATE(,
          ( (UIC),              ,MAKE_UIC),
          ( (NAME)));
!
! Scan for options until end of line is reached
!
        $STATE(OPTIONS,
          ( '/' ),
          ( TPA$_EOS,   TPA$_EXIT ) );
        $STATE(,
          ('OWNER_UIC', PARSE_UIC,      , UIC_FLAG,     PARSER_FLAGS),
          ('ENTRIES',   PARSE_ENTRIES,  , ENTRIES_FLAG, PARSER_FLAGS),
          ('PROTECTION',PARSE_PROT,     , PROT_FLAG,    PARSER_FLAGS) );

!
! Get file owner UIC
!
        $STATE( PARSE_UIC,
          (':'),
          ('='));
        $STATE(,
          ( (UIC),      OPTIONS ));
!
! Get number of directory entries
!
        $STATE( PARSE_ENTRIES,
          (':'),
          ('='));
        $STATE(,
          (TPA$_DECIMAL,OPTIONS,        ,       ,       ENTRY_COUNT));
```

**Figure 8-1 (Cont.): Sample TPARSE Program**

```
!
! Get directory file protection. Note that the bit masks generate the
! protection in complement form. It will be uncomplemented by the main
! program
!
        $STATE( PARSE_PROT,
          (':'),
          ('='));
        $STATE(,
          ('('));

        $STATE( NEXT_PRO,
          ('SYSTEM',    SYPR ),
          ('OWNER',     OWPR ),
          ('GROUP',     GRPR ),
          ('WORLD',     WOPR ) );
        $STATE( SYPR,
          (':'),
          ('=') );

        $STATE(SYPRO,
          ('R',          SYPRO,   , %X'1',        FILE_PROTECT),
          ('W',          SYPRO,   , %X'2',        FILE_PROTECT),
          ('E',          SYPRO,   , %X'4',        FILE_PROTECT),
          ('D',          SYPRO,   , %X'8',        FILE_PROTECT),
          (TPA$_LAMBDA, ENDPRO) );

        $STATE(OWPR,
          (':'),
          ('='));

        $STATE(OWPRO,
          ('R',          OWPRO,   , %X'0010',     FILE_PROTECT),
          ('W',          OWPRO,   , %X'0020',     FILE_PROTECT),
          ('E',          OWPRO,   , %X'0040',     FILE_PROTECT),
          ('D',          OWPRO,   , %X'0080',     FILE_PROTECT),
          (TPA$_LAMBDA, ENDPRO) );

        $STATE(GRPR,
          (':'),
          ('='));

        $STATE(GRPRO,
          ('R',          GRPRO,   , %X'0100',     FILE_PROTECT),
          ('W',          GRPRO,   , %X'0200',     FILE_PROTECT),
          ('E',          GRPRO,   , %X'0400',     FILE_PROTECT),
          ('D',          GRPRO,   , %X'0800',     FILE_PROTECT),
          (TPA$_LAMBDA, ENDPRO) );

        $STATE(WOPR,
          (':'),
          ('='));
```

**Figure 8-1 (Cont.): Sample TPARSE Program**

```
        $STATE(WOPRO,
          ('R',          WOPRO,   , %X'1000',     FILE_PROTECT),
          ('W',          WOPRO,   , %X'2000',     FILE_PROTECT),
          ('E',          WOPRO,   , %X'4000',     FILE_PROTECT),
          ('D',          WOPRO,   , %X'8000',     FILE_PROTECT),
          (TPA$_LAMBDA, ENDPRO) );

        $STATE(ENDPRO,
          (',',          NEXT_PRO),
          (')',          OPTIONS) );

!
! Subexpressions to parse a UIC string.
!
        $STATE( UIC,
          ('[') );
        $STATE(,
          (TPA$_OCTAL,          ,       ,       ,       UIC_GROUP) );
        $STATE(,
          (','));
        $STATE(,
          (TPA$_OCTAL,          ,       ,       ,       UIC_MEMBER) );
        $STATE(,
          (']',          TPA$_EXIT, CHECK_UIC) );
!
! Subexpressions to parse a general directory string
!
        $STATE( NAME,
          ('[') );
        $STATE( NAMEO,
          (TPA$_STRING,          , STORE_NAME) );
        $STATE(,
          ('.',          NAMEO ),
          (']',          TPA$_EXIT) );

!
! Note absence of $END_STATE macro.
!

%SBTTL  'Parser Action Routines'

ROUTINE BLANKS_OFF=
!
! Shut off explicit blank processing after passing the command name.
!
    BEGIN
    BUILTIN
        AP;
    MAP
        AP:     REF BLOCK[,BYTE];

    AP[TPA$V_BLANKS] = 0;
    1                                   ! Success always.
    END;
```

Figure 8-1 (Cont.): Sample TPARSE Program

```
ROUTINE CHECK_UIC =
!
! Check the UIC for legal value range.
!
    BEGIN
    BUILTIN
        AP;
    MAP
        AP:     REF BLOCK[,BYTE];

    IF .UIC_GROUP<16,16> NEQ 0 OR       ! UIC components are 16 bits only
       .UIC_MEMBER<16,16> NEQ 0
    THEN
        RETURN 0;

    UIC_OWNER<0,16> = .UIC_MEMBER<0,16>;
    UIC_OWNER<16,16> = .UIC_GROUP<0,16>;
    1
    END;

ROUTINE STORE_NAME=
!
! Store a directory name component
!
    BEGIN
    BUILTIN
        AP;
    MAP
        AP:     REF BLOCK[,BYTE];

    IF .NAME_COUNT GEQ 3 THEN RETURN 0;         ! Maximum of 3 components

    DIRNAME[.NAME_COUNT,0,0,32,0] = .AP[TPA$L_TOKENCNT]; ! Store the
    DIRNAME[.NAME_COUNT,1,0,32,0] = .AP[TPA$L_TOKENPTR]; !  descriptor
    NAME_COUNT = .NAME_COUNT + 1;                        ! Set to next
                                                         !  free slot

    IF .AP[TPA$L_TOKENCNT] GTR 9 THEN RETURN 0; ! Name too long

    1
    END;

ROUTINE MAKE_UIC=
    BEGIN
    BUILTIN
        AP;
    MAP
        AP:     REF BLOCK[,BYTE];
    OWN
        UIC_STRING:     VECTOR[CH$ALLOCATION(6)];

    IF  .UIC_GROUP<8,8> NEQ 0 OR                ! Check UIC for byte values,
        .UIC_MEMBER<8,8> NEQ 0                  !  since UIC type directories
    THEN                                        !  are restricted to this form
        RETURN 0;

    DIRNAME[0,1,0,32,0] = UIC_STRING;           ! Point to string buffer
    $FAOL( CTRSTR=UPLIT(6,'!OB!OB'),            ! Convert UIC to octal string
           OUTBUF=DIRNAME,
           PRMLST=UIC_GROUP)
    END;
```

Figure 8-1 (Cont.): Sample TPARSE Program

```
%SBTTL  'Main Program'

ROUTINE CREATE_DIR=
!+
!
! This is the main program of the CREATE/DIRECTORY utility. It gets
! the command line from the command interpreter and parses it with TPARSE.
!
!-
    BEGIN

    !
    ! Call the command interpreter to obtain the command line
    !
    SYS$CLI( REQ_COMMAND, 0, 0 );

    !
    ! Copy the input string descriptor into the TPARSE control block
    ! and call LIB$TPARSE. Note that impure storage is assumed to be zero.
    !
    TPARSE_BLOCK[TPA$L_STRINGCNT] = .REQ_COMMAND[CLI$W_RQSIZE];
    TPARSE_BLOCK[TPA$L_STRINGPTR] = .REQ_COMMAND[CLI$A_RQADDR];

    IF NOT LIB$TPARSE( TPARSE_BLOCK, UFD_STATE, UFD_KEY )
    THEN
        RETURN SS$_ABORT;

    !
    ! Parsing is complete
    !

                                        ! Process the command to create
                                        ! the appropriate directory.


    RETURN SS$_NORMAL
    END;                            ! Return Success to command interpreter
END ELUDOM
```

**Figure 8-1 (Cont.): Sample TPARSE Program**

# APPENDIX A

# SUMMARY OF COMMAND SYNTAX

This appendix summarizes the command syntax, the qualifier defaults, and their abbreviations.

## A.1 COMMAND-LINE SYNTAX

| | |
|---|---|
| compilation-request | $ BLISS bliss-command-line |
| bliss-command-line | { qualifier,... } space input-spec,... |
| qualifier | { output qualifier<br>general qualifier<br>terminal qualifier<br>optimization qualifier<br>source-list qualifier<br>machine-code-list qualifier } |
| space | { blank \| tab } ... |
| input-spec | file-spec+...{qualifier,...} |

## A.2 FILE SPECIFICATION SUMMARY

| | |
|---|---|
| file-spec | { node:: } { dev: } { [dir] } file { .type } { ;ver } |
| node | 1- to 6-character alphanumeric network node name and optionally a 1- to 42-character access control string |
| dev | any logical or physical device name; includes device type, controller designation, and unit number |
| dir | any valid directory or subdirectory name |
| file | 1 to 9 alphanumeric characters |
| type | 1 to 3 alphanumeric characters |
| ver | version number from 1 to 32767 |

## A.3 QUALIFIER SYNTAX

| | |
|---|---|
| output qualifier | { /OBJECT {=file-spec} \| /NOOBJECT<br>/LIST {=file-spec} \| /NOLIST<br>/LIBRARY {=file-spec} \| /NOLIBRARY } |

| | |
|---|---|
| general qualifier | { /TRACEBACK \| /NOTRACEBACK<br>/DEBUG \| /NODEBUG<br>/CODE \| /NOCODE<br>/VARIANT {:value} } |

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

| | |
|---|---|
| optimize qualifier | /OPTIMIZE= { ( optimize-value ,... )<br>optimize-value } |
| optimize-value | { QUICK \| NOQUICK<br>SPEED \| SPACE<br>LEVEL : optimize-level<br>SAFE \| NOSAFE } |
| optimize-level | { 0 \| 1 \| 2 \| 3 } |

| | |
|---|---|
| source-list qualifier | /SOURCE_LIST= { ( source-value ,... )<br>source-value } |
| source-value | { PAGE_SIZE : number-of-lines<br>HEADER \| NOHEADER<br>LIBRARY \| NOLIBRARY<br>REQUIRE \| NOREQUIRE<br>EXPAND_MACROS \| NOEXPAND_MACROS<br>TRACE_MACROS \| NOTRACE_MACROS<br>SOURCE \| NOSOURCE } |
| number-of-lines | { 20 \| 21 \| 22 \| ... } |

| | |
|---|---|
| machine-code-<br>list qualifier | /MACHINE_CODE_LIST= { ( code-value ,... )<br>code-value } |
| code-value | { OBJECT \| NOOBJECT<br>ASSEMBLER \| NOASSEMBLER<br>SYMBOLIC \| NOSYMBOLIC<br>BINARY \| NOBINARY<br>COMMENTARY \| NOCOMMENTARY<br>UNIQUE_NAMES \| NOUNIQUE_NAMES } |

## A.4 QUALIFIER DEFAULTS

The following qualifiers are assumed by default:

```
/OBJECT=input-file-name.OBJ

/NOLIST (in interactive mode)
/LIST   (in batch mode)

/NOLIBRARY

/TRACEBACK

/ERROR_LIMIT=30

/NODEBUG

/CODE

/VARIANT:0

/TERMINAL=(ERRORS,NOSTATISTICS)

/OPTIMIZE=(NOQUICK,SPACE,LEVEL:2,SAFE)

/SOURCE_LIST=(HEADER,PAGE_SIZE:58,NOLIBRARY,NOREQUIRE,
              NOEXPAND_MACROS,NOTRACE_MACROS,SOURCE)

/MACHINE_CODE_LIST=(NOASSEMBLER,SYMBOLIC,BINARY,OBJECT
                    COMMENTARY,NOUNIQUE_NAMES)
```

## A.5 ABBREVIATIONS

The abbreviations for the positive forms of the qualifiers and qualifier values are given below:

| Qualifier | Value | Abbreviation |
|---|---|---|
| /OBJECT | | /OB |
| /LIST | | /LIS |
| /LIBRARY | | /LIB |
| /ERROR_LIMIT | | /E |
| /TRACEBACK | | /TR |
| /VARIANT | | /V |
| /CODE | | /C |
| /DEBUG | | /D |
| /TERMINAL | | /TE |
| | ERRORS | E |
| | STATISTICS | S |
| /OPTIMIZE | | /OP |
| | QUICK | Q |
| | SPACE | SPA |
| | SPEED | SPE |
| | LEVEL | L |
| | SAFE | SA |

| Qualifier | Value | Abbreviation |
|---|---|---|
| /SOURCE_LIST | | /S |
| | HEADER | H |
| | PAGE_SIZE | P |
| | LIBRARY | L |
| | REQUIRE | R |
| | EXPAND_MACROS | E |
| | TRACE_MACROS | T |
| | SOURCE | S |
| /MACHINE_CODE_LIST | | /M |
| | OBJECT | O |
| | ASSEMBLER | A |
| | SYMBOLIC | S |
| | BINARY | B |
| | COMMENTARY | C |
| | UNIQUE_NAMES | U |

For the negative form of a qualifier or value (where applicable), its positive-form abbreviation can be prefixed by "NO".

## 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 2 are indented correctly, although all comments have been omitted in order to save space.

For further information, see the VAX-11 Software Engineering Manual.

# 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 (Page C-2) appears first. It provides documentation explaining the module's function, use, and history.

The module's declarative part appears next (Page C-3). 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, consisting of zero or more routines, comes next. The template for a routine is on Page C-4. A routine has three parts: a preface, a declarative part, and code.

Finally, every module has a closing part (Page 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 template and its use are described in detail in the VAX-11 Software Engineering Manual.

The file MODULE.BLI is supplied as part of the BLISS support package, on logical device SYS$LIBRARY.

## C.1 MODULE PREFACE

```
MODULE TEMPLATE (       !
                IDENT = ' '
                ) =
BEGIN

!
!             COPYRIGHT (C) 1978, 1979, 1980 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-32 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-32 LANGUAGE

| The maximum number of | is |
|---|---|
| • Nested blocks containing declarations | 64 |
| • Characters in a quoted-string | 1000 |
| • Actual parameters in a routine call | 64 |
| • Structure formal parameters | 31 |
| • Field components | 32 |
| • Parameters of a FIELD attribute | 128 |
| • Bytes initialized by a single PLIT (i.e., the maximum byte count of a single PLIT) | 65,535 |

## D.2 SYSTEM INTERFACES

| The maximum number of | is |
|---|---|
| • Characters in an input source line | 132 |
| • 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 closed parenthesis has been encountered when the compiler expected an END.

**User Action:** Provide the appropriate pairing or insert a missing END keyword.

004 Missing operand preceding "<operator-name>"

**Explanation:** Required left-operand is missing from infix-operator named.

**User Action:** Insert missing left-operand.

005 Control expression must be parenthesized

**Explanation:** Parenthesis is required to achieve intended result.

**User Action:** Insert missing parenthesis.

006 Superfluous operand following "<operator-name>"

**Explanation:** An extra or unecessary right-operand follows the operator named.

**User Action:** Remove the excess or unecessary right-operand.

007 Missing operand following "<operator-name>"

**Explanation:** Required right-operand is missing from operator named.

**User Action:** Insert missing right-operand.

008 Missing THEN following IF

**Explanation:** Conditional-expression is incomplete.

**User Action:** Insert required keyword THEN.

009 Missing DO following WHILE or UNTIL

**Explanation:** Pre-tested-loop-expression is incomplete.

**User Action:** Insert required keyword DO.

010 Missing WHILE or UNTIL following DO

**Explanation:** Post-tested-loop-expression is incomplete.

**User Action:** Insert required keyword WHILE or UNTIL.

011 Name longer than 31 characters

**Explanation:** Maximum name length has been exceeded.

**User Action:** Reduce name length to 31 characters or less.

012 Missing DO following INCR or DECR

**Explanation:** Indexed-loop-expression is incomplete.

**User Action:** Insert required keyword DO.

013 Missing comma or right parenthesis in routine actual parameter list

**Explanation:** Each actual-parameter in a list must be separated by a comma and the list must be ended by a close parenthesis.

**User Action:** Insert 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 semicolon or keyword TES following select-action expression.

021 Address arithmetic involving REGISTER variable <variable-name>

**Explanation:** An attempt has been made to use the value of a register name in an expression.

**User Action:** Correct the expression.

022 Field reference used as an expression has no value

**Explanation:** The reference is invalid as a fetch or assignment expression and cannot produce a value.

**User Action:** Evaluate and validate the expression.

023 Missing comma or right angle bracket in a field selector

**Explanation:** Field selector is incomplete.

**User Action:** Insert missing 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: An error has occurred in the coding of an attribute.

**User Action:** Correct the error using the appropriate syntax.

034 INITIAL value <integer> too large for field

**Explanation:** The integer value shown is too large for the designated field.

**User Action:** Decrease the value or increase the allocation-unit.

035 The <attribute-name> attribute contradicts corresponding FORWARD declaration

**Explanation:** The attributes of a name in an own- 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.

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 that was compiled with the assembleable-listing option.

**User Action:** If the intent is to run the listing through an assembler, delete all extra appearances of the declared name; or, use the switch-item UNAMES in a switches-declaration to obtain unique names.

193 <declaration-name> declaration not available when OBJECT(ABSOLUTE) in effect

**Explanation:** This message is reserved for BLISS-16 future expansion.

**User Action:** No action is required.

194 Library source module must contain only declarations

**Explanation:** Executable expressions must not appear in a library source file.

**User Action:** Remove all but 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: <keyword>

**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 interupt 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 paramete-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.

## 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 1024 (VAX) or 2048 (10/20) disk blocks; the library file will be deleted.

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 2 examples use excerpts from this listing.

```
TESTFACT                                    5-Oct-1979 10:37:06    VAX-11 Bliss-32 T2-617              Page   1
                                            5-Oct-1979 10:37:00    DBB1:[LEHOTSKY.DOC.UGUIDE]TESTFACT.BLI;2 (1)

;          0001  MODULE TESTFACT (MAIN = MAINPROG)=
;          0002  BEGIN
;          0003
;          0004  OWN
;          0005      A,
;          0006      B,
;          0007      C;
;          0008
;          0009  ROUTINE IFACT (N) =
;          0010      BEGIN
;          0011
;          0012      LOCAL
;          0013          RESULT;
;          0014      RESULT = 1;
;          0015      INCR I FROM 2 TO .N DO
;          0016          RESULT = .REULT*.I;
; WARN#000      .................1  L1:0016
; Undeclared name:  REULT
;          0017      .RESULT
;          0018      END;

                                                                .TITLE  TESTFACT

                                                                .PSECT  $OWN$,NOEXE,2

                                                    00000 A:    .BLKB   4
                                                    00004 B:    .BLKB   4
                                                    00008 C:    .BLKB   4

                                                                .EXTRN  REULT

                                                                .PSECT  $CODE$,NOWRT,2

                                                0000 00000 IFACT:   .WORD   Save nothing                    ; 0009
                                    50              01  D0 00002         MOVL    #1, RESULT              ; 0014
                                    51              01  D0 00005         MOVL    #1, I                   ; 0015
                                                    06  11 00008         BRB     2$                      ;
                          50    0000G   CF          51  C5 0000A 1$:     MULL3   I, REULT, RESULT        ; 0016
                          F5            51    04    AC  F3 00010 2$:     AOBLEQ  N, I, 1$                ; 0015
                                                        04 00015         RET                             ; 0009

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


;          0019
```

Figure F-1: Sample Output Listing

```
TESTFACT                                    5-Oct-1979 10:37:06    VAX-11 Bliss-32 T2-617              Page   2
                                            5-Oct-1979 10:37:00    DBB1:[LEHOTSKY.DOC.UGUIDE]TESTFACT.BLI;2 (2)

;          0020
;          0021  ROUTINE RFACT (N) =
;          0022      IF .N GTR 1
;          0023      THEN
;          0024          .N * RFACT(.N - 1)
;          0025      ELSE
;          0026          1;


                                                0000 00000 RFACT:  .WORD   Save nothing                    ; 0021
                                 01       04    AC  D1 00002       CMPL    N, #1                         ; 0022
                                                0E  15 00006       BLEQ    1$                            ;
                           7E    04       AC        01  C3 00008   SUBL3   #1, N, -(SP)                  ; 0024
                                 EF       AF        01  FB 0000D   CALLS   #1, RFACT                     ;
                                 50       04    AC  C4 00011       MULL2   N, R0                         ;
                                                    04 00015       RET                                   ; 0022
                                 50             01  D0 00016 1$:   MOVL    #1, R0                        ;
                                                    04 00019       RET                                   ; 0021

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

Figure F-1 (Cont.): Sample Output Listing

```
TESTFACT                                                5-Oct-1979 10:37:06    VAX-11 Bliss-32 T2-617              Page   3
                                                        5-Oct-1979 10:37:00    DBB1:[LEHOTSKY.DOC.UGUIDE]TESTFACT.BLI;2 (3)

;          0027  ROUTINE MAINPROG =
;          0028      BEGIN
;          0029      A = IFACT(5);
;          0030      B = RFACT(5);
;          0031
;          0032      1                           ! VMS wants a success return
;          0033      END;


                                                      0000 00000 MAINPROG:
                                                                      .WORD   Save nothing                            ; 0027
                                                           05  DD 00002      PUSHL   #5                              ; 0029
                                                C8    AF   01  FB 00004      CALLS   #1, IFACT                       ;
                                              0000'   CF   50  D0 00008      MOVL    R0, A                           ;
                                                           05  DD 0000D      PUSHL   #5                              ; 0030
                                                D3    AF   01  FB 0000F      CALLS   #1, RFACT                       ;
                                              0000'   CF   50  D0 00013      MOVL    R0, B                           ;
                                                      50   01  D0 00018      MOVL    #1, R0                          ; 0027
                                                           04  0001B      RET                                     ;

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


;          0034  END ELUDOM
```

Figure F-1 (Cont.): Sample Output Listing

```
;                                       PSECT SUMMARY
;
;       Name                            Bytes                           Attributes
;
;  $OWN$                                  12    WRT,  RD ,NOEXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)
;  $CODE$                                 76  NOWRT,  RD ,  EXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)



; Warnings:      1
; Errors:        0


;                                       COMMAND QUALIFIERS

;       BLISS /LIS/NOOB TESTFACT

; Size:          76 code + 12 data bytes
; Run Time:      00:01.2
; Elapsed Time:  00:04.1
; Memory Used:   10 pages
; Compilation Complete
```

Figure F-1 (Cont.): Sample Output Listing

# INDEX

-T-

VAX-11 BLISS-32
User's Guide
AA-H322C-TE

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 — — — — — — — — — — — — — — — — — — — — — —