BLISS

Pocket Guide

BLISS

digital

This pocket guide applies to the following BLISS compilers:

BLISS-16C	Version 4
BLISS-32	Version 1
BLISS-36	Version 1A

Order No. AV-H289A-TE

IMPORTANT NOTICE

Unlike BLISS-32, the BLISS-16C and BLISS-36 compilers are internal tools developed by DIGITAL for its own use. They are not available as products of Digital Equipment Corporation.

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.

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

Printed in U.S.A.

TABLE OF CONTENTS

SYNTAX SUMMARY

COMMAND SUMMARY

SCOPE AND INTENT OF THIS GUIDE

This guide presents a syntax summary for the family of BLISS language dialects consisting of BLISS-16C, BLISS-32, and BLISS-36. It describes the Common BLISS Language features that constitute the bulk of all three dialects, plus the additional system-specific features unique to each dialect. A summary of the command-line syntax for the respective compilers is also provided.

The guide is primarily intended as a concise syntax reference for knowledgeable users of BLISS. (It also serves as a convenient means of comparing the several dialects.)

As an additional feature, the guide briefly describes the principal characteristics of BLISS and presents a short sample program, for the benefit of readers with no prior knowledge of the language.

PRESENTATION CONVENTIONS

Common BLISS Versus Dialect Syntax: All Common BLISS syntax is printed in black; all variant syntax (and any commentary associated with it) is printed in color. A numeric flag (16, 32, or 36) at the left margin of the page is used to indicate the dialect(s) to which the variant syntax belongs.

Method of Definition: The general method of syntax definition is the same as that employed in other BLISS language documentation. A language construct definition consists of a set of production rules. Each rule defines a syntactic name (a descriptive name for a meaningful "piece" of the language), often in terms of other, lower-level syntactic names.

Syntax Notation: Each production rule consists of a production name and the symbol → followed by a replacement for the production name, indented on a new line. Production names (always syntactic names) are lower-case words of two or more letters, hyphenated if multiword (for example, control-expression). Production-name replacements may be other syntactic names, syntactic literals, or a combination of the two. For example:

```
tested-loop-expression →
        DO exp { WHILE | UNTIL } exp
```

Syntactic literals consist of all character strings that are neither syntactic names nor the notational symbols defined below.

The notational symbols and conventions used are as follows:

- The vertical-bar symbol (|) separates multiple alternatives listed on one line; otherwise, alternatives are listed vertically (at a uniform level of indentation; see Line Folding below).

- Braces, { }, enclose a set of alternatives, of which one and only one is to be selected, or enclose a single optional construct.

- The symbol " , . . . " denotes an optional repetition of the item immediately preceding, with successive instances separated by ",".

- The symbol " . . . " denotes an optional repetition of the item immediately preceding, with no separating delimiter.

- The symbol "———" indicates omission of part of an ordered sequence of alternatives, such as the alphabet or the numerals 0 through 9.

- Line Folding: A replacement element appearing on a new line that is *further indented* than the preceding line indicates a continuation of the preceding replacement element, rather than an alternative to it. (Such "line folding" is dictated by line-length constraints.)

- Defaults: Unconditional keyword defaults are indicated by underlining of the default keyword.

Abbreviations: The following abbreviations are used:

exp	for	expression
ctce	for	compile-time-constant-expression
ltce	for	link-time-constant-expression

Note particularly that "exp" and "expression" are used interchangeably (due to format constraints).

Also, any syntactic name that ends in "-name" or "-exp" represents a name or expression, respectively, and is not further defined.

Use of Italics: A few quite obvious deviations from the standard presentation format occur, primarily to allow for semantic comments. Such comments are always in italics.

PRINCIPAL CHARACTERISTICS OF BLISS

BLISS is a language designed for building system software. It provides the higher-level language features that are desirable for that purpose, and omits those that introduce inefficiency or a degree of complexity inconsistent with its performance and transportability goals. BLISS also provides facilities for accessing specific hardware functions, yet does so in a manner that clearly distinguishes between machine-independent code (transportable Common BLISS) and the elective machine-specific functions. Overall, it is best characterized as a "medium level" language.

The principal characteristics of BLISS that differ from most widely known languages are as follows:

- All constructs of the BLISS language except declarations are forms of expressions. Statements which perform actions without producing values, do not exist in BLISS. Whenever a BLISS expression is used in a statement-like way, it must be terminated by a semicolon. The compiler will then discard its value.

- The name of a storage location always represents the address of that storage location. Thus, address arithmetic can be accomplished in a simple and consistent fashion. When the contents of a storage location is needed, a fetch operator (.) must prefix the name of the storage location.

- BLISS is a "typeless" language, that is, the type of a given data item is not declared and is not an attribute of the item. The interpretation of the value of a constant or variable depends upon the operator that is applied to it.

- A value is assigned to a storage location by means of the normal assignment operator, "=". However, there is no restriction on the operand that appears on the left-hand side of the assignment. That is, the storage location operand can be any expression yielding an address value.

- The familiar GOTO construct is excluded from BLISS because it permits unclear and unreliable patterns of control flow. Equally important, "GOTO-less" programs are more amenable to global flow optimization. BLISS provides control expressions that are inherently more effective, including IF, CASE, and WHILE.

SAMPLE COMMON BLISS PROGRAM

Note that this program calls on the "EZIO" character-string I/O package for basic file and terminal I/O services.

```
MODULE LISTER ( MAIN = LSTR ) =

BEGIN
!+
! This program asks for a file name, opens the named
! file, and copies the file to the terminal.
!-

EXTERNAL ROUTINE
    FILOPN,                     ! Ezio open
    FILCLS,                     ! Ezio close
    FILOUT,                     ! Ezio output
    FILIN;                      ! Ezio input

OWN             ! Holds one line of text.
    BUF : VECTOR[CH$ALLOCATION(120)];

MACRO           ! Outputs literal string to tty.
    MSG (S) =
        FILOUT(-1,%CHARCOUNT(S),CH$PTR(UPLIT(S))) %;


ROUTINE LSTR =
    BEGIN

    LOCAL
        LEN,                    ! Length of the string.
        PTR;                    ! Pointer to buf.

    ! Open the tty.  Note: no filespec.
    FILOPN(-1, 0, 0, 0);
    PTR = CH$PTR(BUF);          ! Get pointer.
    MSG('ENTER FILE NAME: ');   ! Prompt.
    LEN = FILIN(-1, 60, .PTR);  ! Get file name.

    ! Open the file on channel 0.
    IF NOT FILOPN(0, .LEN, .PTR, 0)
    THEN
        BEGIN                   ! Open failed.
        MSG('OPEN FAILED.');
        RETURN
        END;

    ! Process each line
    WHILE 1 DO
        BEGIN
        LEN = FILIN(0, 120, .PTR);

        IF .LEN EQL -1
        THEN
            EXITLOOP;           ! End of file.

        FILOUT(-1, .LEN, .PTR)  ! Output the string.
        END;

    FILCLS(0);                  ! Close the input file.
    MSG('DONE.')                ! A message.
    END;

END
ELUDOM
```

SYNTAX SUMMARY

1.0 MODULES

```
module →
        MODULE module-head =
           module-body
           ELUDOM

module-head →
        name { ( module-switch , . . . ) }

module-body →
       { BEGIN declaration . . . END }
       { ( declaration . . . )       }
```

declaration — *See Section 4.0.*

1.1 Module Switches

```
module-switch →
        { on-off-switch | special-switch }

on-off-switch →
       { CODE | NOCODE              }
       { DEBUG | NODEBUG            }
       { ERRS | NOERRS              }
       { OPTIMIZE | NOOPTIMIZE      }
       { UNAMES | NOUNAMES          }
       { SAFE | NOSAFE              }
       { ZIP | NOZIP                }

special-switch →
          { IDENT = quoted-string                          }
          { LANGUAGE ( language-list )                     }
          { LINKAGE ( linkage-name )                       }
          { LIST ( list-option , . . . )                   }
          { MAIN = routine-name                            }
          { OPTLEVEL = { 0 | 1 | 2 | 3 }                   }
          { VERSION = quoted-string                        }
16        { ADDRESSING_MODE ( mode-16 )                    }
32        { ADDRESSING_MODE ( mode-spec , . . . )          }
36   [    { ENTRY ( global-name , . . . )                  }
     [    { ENVIRONMENT ( environ-option , . . . )         }
16        { OBJECT ( object-option )                       }
36        { OTS = quoted-string                            }
```

1.0 MODULES, Continued

```
language-list →
    ⎧ COMMON              ⎫
    ⎨ language-name , . . . ⎬
    ⎩ nothing             ⎭

language-name →
    {BLISS16 | BLISS32 | BLISS36}
```

Note: The effective default is 'no checking'.

```
list-option →
    ⎧ SOURCE     | NOSOURCE       ⎫
    ⎪ REQUIRE    | NOREQUIRE      ⎪
    ⎪ EXPAND     | NOEXPAND       ⎪
    ⎪ TRACE      | NOTRACE        ⎪
    ⎨ LIBRARY    | NOLIBRARY      ⎬
    ⎪ OBJECT     | NOOBJECT       ⎪
    ⎪ ASSEMBLY   | NOASSEMBLY     ⎪
    ⎪ SYMBOLIC   | NOSYMBOLIC     ⎪
    ⎪ BINARY     | NOBINARY       ⎪
    ⎩ COMMENTARY | NOCOMMENTARY   ⎭

16  mode-16 →
        {ABSOLUTE | RELATIVE}

32  mode-spec →
        ⎧ EXTERNAL = mode-32    ⎫
        ⎩ NONEXTERNAL = mode-32 ⎭

32  mode-32 →
        ⎧ ABSOLUTE      ⎫
        ⎨ GENERAL       ⎬
        ⎪ LONG_RELATIVE ⎪
        ⎩ WORD_RELATIVE ⎭

16  object-option →
        {ABSOLUTE | RELOCATABLE}

36  environ-option →
        ⎧ BLISS10_OTS | BLISS36C_OTS ⎫
        ⎪ KA10 | KI10 | KL10         ⎪
        ⎨ EXTENDED                   ⎬
        ⎪ STACK = segment-name       ⎪
        ⎩ TOPS10 | TOPS20            ⎭
```

2.0 EXPRESSIONS

```
expression →
    ⎧ primary             ⎫
    ⎨ operator-expression ⎬
    ⎪ executable-function ⎪
    ⎩ control-expression  ⎭
```

2.1 Primaries

```
primary →
    ⎧ numeric-literal     ⎫
    ⎪ string-literal      ⎪
    ⎪ plit                ⎪
    ⎨ name                ⎬
    ⎪ block               ⎪
    ⎪ structure-reference ⎪
    ⎪ routine-call        ⎪
    ⎩ codecomment         ⎭
```

2.1.1 Numeric Literals

```
numeric-literal →
        ⎧ decimal-literal        ⎫
        ⎨ integer-literal        ⎬
        ⎪ character-code-literal ⎪
32      ⎩ float-literal          ⎭

decimal-literal →
        opt-sign  decimal-digit . . .

opt-sign →
        {+ | − | nothing}

decimal-digit →
        {0 | 1 | 2 | ––– | 8 | 9}

integer-literal →
        {%B | %O | %DECIMAL | %X}
              ' opt-sign  integer-digit . . . '

integer-digit →
        ⎧ 0 | 1 | ––– | 9         ⎫
        ⎩ A | B | C | D | E | F   ⎭

character-code-literal →
        %C ' quoted-character '

32  float-literal →
        ⎧ %E 'mantissa {E exponent} ' ⎫
        ⎩ %D 'mantissa {D exponent} ' ⎭
```

2.1.2 String Literals

```
string-literal →
         {string-type} quoted-string

quoted-string →
          ' quoted-character . . . '

quoted-character →
        ⎧ printing-char-except-apostrophe ⎫
        ⎨ blank                           ⎬
        ⎪ tab                             ⎪
        ⎩ ' '                             ⎭
```

Note: Two consecutive apostrophe characters represent a single apostrophe within a quoted-string.

```
         string-type →
                  ⎧ %ASCII     | %ASCIZ  ⎫
16, 32            ⎨ %RAD50_11  | %ASCIC  ⎬
    36            ⎪ %RAD50_10  | %SIXBIT ⎪
    32            ⎩ %P                   ⎭
```

2.1.3 Plits

```
         plit →
               {PLIT | UPLIT}
16, 32             {alloc-unit}
                    ( plit-item , . . . )

         plit-item →
              ⎧ plit-group      ⎫
              ⎨ plit-expression ⎬
              ⎩ plit-string     ⎭

         plit-group →
              ⎧ alloc-unit                     ⎫
16, 32        ⎨ REP replicator OF              ⎬
16, 32        ⎩ REP replicator OF alloc-unit   ⎭
                 ( plit-item , . . . )

         replicator →
               ctce

         plit-expression →
               ltce

         plit-string →
               string-literal
```

```
         alloc-unit →
              ⎡   ⎧ BYTE ⎫
16, 32        ⎢   ⎨ WORD ⎬
    32        ⎣   ⎩ LONG ⎭
```

Default for 16: WORD; for 32: LONG.

2.1.4 Names

```
name →
     {letter | dollar | underscore}
        ⎧ letter | dollar | underscore ⎫
        ⎩ digit | nothing              ⎭ . . .

letter →
     ⎧ A | B | C | ––– | Z ⎫
     ⎩ a | b | c | ––– | z ⎭

dollar →
       $

underscore →
       _

digit →
     {0 | 1 | 2 | ––– | 9}
```

Note: A name may not contain more than 15 characters.

2.1.5 Blocks

```
block →
     ⎧ labeled-block   ⎫
     ⎩ unlabeled-block ⎭

labeled-block →
     attached-label . . . unlabeled-block

attached-label →
     label-name :

unlabeled-block →
     ⎧ BEGIN block-body END ⎫
     ⎩ ( block-body )       ⎭
```

```
block-body →
        {declaration . . .}
           {block-action . . .}
           {block-value}
```

Note: The block-body must not be null.

```
block-action →
        expression ;
```

```
block-value →
        expression
```

2.1.6 Structure References

```
structure-reference →
     ⎧ordinary-structure-reference⎫
     ⎩general-structure-reference ⎭
```

```
ordinary-structure-reference →
        segment-name [ {access-actual , . . .} ]
```

```
segment-name →
        name   — of a data-segment declared with a
                 structure attribute; see Section 4.2.
```

```
access-actual →
     {exp | field-name | nothing}
```

field-name — *See field-attribute, Section 4.1.*

```
general-structure-reference →
        structure-name [ access-part
          {; alloc-actual , . . .} ]
```

```
structure-name →
        name   — of user-declared or predeclared
                 structure; see Sections 4.3 and 6.3.
```

```
access-part →
     {segment-expression}
        {, access-actual , . . .}
```

alloc-actual — *See structure-attribute, Section 4.1.*

2.1.7 Routine Calls

```
routine-call →
     ⎧ordinary-routine-call⎫
     ⎩general-routine-call ⎭
```

```
ordinary-routine-call →
        routine-designator
          ( {actual-parameter , . . .} )
```

```
routine-designator →
        primary
```

```
actual-parameter →
        expression
```

```
general-routine-call →
        linkage-name ( routine-address
          {, actual-parameter , . . .} )
```

linkage-name — *See Sections 4.6 and 6.4 for a summary of linkage names.*

```
routine-address →
        expression
```

2.1.8 Codecomments

```
codecomment →
        CODECOMMENT quoted-string , . . . :
          block
```

2.2 Operator Expressions

```
operator-expression →
     ⎧fetch-expression ⎫
     ⎪prefix-expression⎪
     ⎨infix-expression ⎬
     ⎩assign-expression⎭
```

```
fetch-expression →
        primary {field-selector}
```

field-selector — *See Section 2.2.1.*

```
prefix-expression →
        {+ | - | NOT} op-exp

op-exp →
       ⎧ primary             ⎫
       ⎨ operator-expression ⎬
       ⎩ executable-function ⎭

executable-function     – See Section 2.3.

infix-expression →
        op-exp  infix-operator  op-exp

infix-operator →
       ⎧ +    | -    | *    ⎫
       ⎪ /    | MOD  | ^    ⎪
       ⎪ EQL  | EQLA | EQLU ⎪
       ⎪ NEQ  | NEQA | NEQU ⎪
       ⎪ LSS  | LSSA | LSSU ⎪
       ⎨ LEQ  | LEQA | LEQU ⎬
       ⎪ GTR  | GTRA | GTRU ⎪
       ⎪ GEQ  | GEQA | GEQU ⎪
       ⎪ AND  | OR          ⎪
       ⎩ EQV  | XOR         ⎭

assign-expression →
        op-exp  {field-selector} = op-exp
```

2.2.1 Field Selectors

```
field-selector →
        < position-exp, size-exp {,sign-ext-flag} >

sign-ext-flag →
        ctce     – Value:  0 or 1
```

Note: The permissible value range for position-exp (p) and size-exp (s) is as follows:

BLISS-16	BLISS-32	BLISS-36
$0 \leqslant p$ $p + s \leqslant 16$ $0 \leqslant s \leqslant 16$	$0 \leqslant s \leqslant 32$	$0 \leqslant p$ $p + s \leqslant 36$ $0 \leqslant s \leqslant 36$

2.2.2 Operator Precedence

The operator-expressions are listed in the following table in order of decreasing priority level, with an associativity for the operators at each level. (Abbreviations: exp1 and exp2 represent any op-exp as defined in Section 2.2; "R" stands for right and "L" for left.)

Priority	Operator Expression	Associates from
highest	fetch-expression	R to L
	$\left\{\begin{matrix} + \\ - \end{matrix}\right\}$ exp2	R to L
	exp1 ^ exp2	L to R
	exp1 $\left\{\begin{matrix} \text{MOD} \\ * \\ / \end{matrix}\right\}$ exp2	L to R
	exp1 $\left\{\begin{matrix} + \\ - \end{matrix}\right\}$ exp2	L to R
	exp1 $\left\{\begin{matrix} \text{EQLx} \\ \text{NEQx} \\ \text{LSSx} \\ \text{LEQx} \\ \text{GTRx} \\ \text{GEQx} \end{matrix}\right\}$ exp2	L to R
	NOT exp2	R to L
	exp1 AND exp2	L to R
	exp1 OR exp2	L to R
	exp1 $\left\{\begin{matrix} \text{EQV} \\ \text{XOR} \end{matrix}\right\}$ exp2	L to R
lowest	assign-expression	R to L

2.3 Executable Functions

```
executable-function →
        executable-function-name
            ( {actual-parameter , . . . } )

executable-function-name →
        ⎧ standard-function-name         ⎫
        ⎪ linkage-function-name          ⎪
        ⎨ supplementary-function-name    ⎬
        ⎪ machine-specific-function-name ⎪
16, 32  ⎩ cond-handling-function-name    ⎭

actual-parameter →
        expression
```

2.3.1 Function Names

standard-function-name →

The names and syntax of the standard functions, plus brief semantic descriptions of each, are as follows:

ABS (e1)	—	*Absolute value of signed integer*
MAX (e1, e2 , . . .)	—	*Maximum of signed integer set*
MAXA (e1, e2 , . . .)	—	*Maximum of address-value set*
MAXU (e1, e2 , . . .)	—	*Maximum of unsigned integer set*
MIN (e1, e2 , . . .)	—	*Minimum of signed integer set*
MINA (e1, e2 , . . .)	—	*Minimum of address-value set*
MINU (e1, e2 , . . .)	—	*Minimum of unsigned integer set*
SIGN (e1)	—	*Sign of a signed integer value*
%REF (e1)	—	*Temporary address of actual parameter*

where e1 and e2 represent expressions.

linkage-function-name →

— *See Section 6.4 for a summary of the linkage functions.*

supplementary-function-name →

— *See Section 6.5 for a summary of the supplementary functions.*

machine-specific-function-name →

— *See Section 7.0 for a summary of the machine-specific function names.*

```
16, 32  cond-handling-function-name →
            ⎧ SIGNAL      ⎫
            ⎨ SIGNAL_STOP ⎬
            ⎩ SETUNWIND   ⎭
```

2.4 Control Expressions

```
control-expression →
        ⎧ conditional-expression ⎫
        ⎪ case-expression        ⎪
        ⎨ select-expression      ⎬
        ⎪ loop-expression        ⎪
        ⎪ exit-expression        ⎪
        ⎩ return-expression      ⎭
```

2.4.1 Conditional Expressions

```
conditional-expression →
        IF exp THEN exp {ELSE exp}
```

2.4.2 Case Expressions

```
case-expression →
        CASE exp
            FROM ctce TO ctce OF
            SET
            case-line . . .
            TES

case-line →
        [ case-label , . . . ]  :  case-action ;

case-label →
        ⎧ ctce         ⎫
        ⎨ ctce TO ctce ⎬
        ⎪ INRANGE      ⎪
        ⎩ OUTRANGE     ⎭

case-action →
        expression
```

2.4.3 Select Expressions

```
select-expression →
        select-type
           select-index OF
           SET
           select-line . . .
           TES

select-type →
      { SELECT    | SELECTA    | SELECTU    }
      { SELECTONE | SELECTONEA | SELECTONEU }

select-index →
        expression

select-line →
        [ select-label , . . . ]  :  select-action ;

select-label →
      { exp        }
      { exp TO exp }
      { OTHERWISE  }
      { ALWAYS     }

select-action →
        expression
```

2.4.4 Loop Expressions

```
loop-expression →
      { indexed-loop-expression }
      { tested-loop-expression  }

indexed-loop-expression →
        index-loop-type  name
           {FROM exp}{TO exp}
           {BY exp}
            DO exp

index-loop-type →
      { INCR | INCRA | INCRU }
      { DECR | DECRA | DECRU }
```

```
tested-loop-expression →
      { pre-tested-loop  }
      { post-tested-loop }

pre-tested-loop →
      {WHILE | UNTIL} exp DO exp

post-tested-loop →
        DO exp {WHILE | UNTIL} exp
```

2.4.5 Exit Expressions

```
exit-expression →
      { leave-expression    }
      { exitloop-expression }

leave-expression →
        LEAVE  label-name {WITH exp}

exitloop-expression
        EXITLOOP {exp}
```

2.4.6 Return Expressions

```
return-expression →
        RETURN {exp}
```

3.0 CONSTANT EXPRESSIONS

3.1 Compile-Time Constant Expressions (ctce)

ctce →

— is any constant expression that can be evaluated during compilation of the module in which it appears.

3.2 Link-Time Constant Expressions (ltce)

ltce →

— is any constant expression that can be evaluated by the time the module is bound into executable form by the linker.

4.0 DECLARATIONS

```
declaration →
           { data-declaration          }
           { structure-declaration     }
           { field-declaration         }
           { routine-declaration       }
           { linkage-declaration       }
16, 32     { enable-declaration        }
           { bound-declaration         }
           { compiletime-declaration   }
           { macro-declaration         }
           { require-declaration       }
           { library-declaration       }
16, 32     { psect-declaration         }
           { switches-declaration      }
           { label-declaration         }
           { builtin-declaration       }
           { undeclare-declaration     }
```

4.1 Common Declaration Attributes

The following attributes are either common to many of the declarations named above or have a fairly complex syntax structure (or both):

```
            structure-attribute
            field-attribute
16, 32 [    allocation-unit
       [    extension-attribute
32          addressing-mode-attribute
```

These attributes are defined immediately below, prior to the individual declaration descriptions. All other attributes are defined in the declaration descriptions themselves.

The Structure Attribute

```
structure-attribute →
     {REF} structure-name
          {[alloc-actual , . . . ] }
```

structure-name — *Either user-declared or predeclared; see Sections 4.3 and 6.3.*

```
alloc-actual →
           { ctce                }
16, 32 [   { allocation-unit     }
       [   { extension-attribute }
           { nothing             }
```

The Field Attribute

```
field-attribute →
     FIELD ( { field-name     } , . . . )
             { field-set-name }
```

Note: See Section 4.4 for definition of field-name and field-set-name.

The Allocation Unit

```
allocation-unit →
16, 32 [   { BYTE }
       [   { WORD }
32         { LONG }
```

Default for 16: WORD; for 32: LONG

The Extension Attribute

```
16, 32 [ extension-attribute →
       [      {SIGNED | UNSIGNED}
```

The Addressing-Mode Attribute

```
   [ addressing-mode-attribute →
   [      ADDRESSING_MODE ( mode-32 )
   [
   [ mode-32 →
32 [      { ABSOLUTE      }
   [      { GENERAL       }
   [      { LONG_RELATIVE }
   [      { WORD_RELATIVE }
```

4.2 Data Declarations

```
        data-declaration →
                ⎧ own-declaration                  ⎫
                ⎪ global-declaration               ⎪
                ⎪ external-declaration             ⎪
32, 36          ⎪ forward-declaration              ⎪
                ⎨ local-declaration                ⎬
                ⎪ stacklocal-declaration           ⎪
                ⎪ register-declaration             ⎪
32, 36          ⎪ global-register-declaration      ⎪
                ⎪ external-register-declaration    ⎪
                ⎩ map-declaration                  ⎭
```

4.2.1 Own Declarations

```
        own-declaration →
                OWN own-item , . . . ;

        own-item →
                own-name {: own-attribute . . .}

        own-attribute →
                ⎧ structure-attribute              ⎫
                ⎪ field-attribute                  ⎪
                ⎪ INITIAL ( plit-item , . . . )    ⎪
16, 32          ⎨ allocation-unit                  ⎬
                ⎪ extension-attribute              ⎪
32              ⎪ ALIGN ( boundary-ctce )          ⎪
                ⎩ VOLATILE                         ⎭
```

4.2.2 Global Declarations

```
        global-declaration →
                GLOBAL global-item , . . . ;

        global-item →
                global-name {: global-attribute . . .}

        global-attribute →
                ⎧ structure-attribute              ⎫
                ⎪ field-attribute                  ⎪
                ⎪ INITIAL ( plit-item , . . . )    ⎪
16, 32          ⎨ allocation-unit                  ⎬
                ⎪ extension-attribute              ⎪
32              ⎪ ALIGN ( boundary-ctce )          ⎪
                ⎪ VOLATILE                         ⎪
32              ⎩ WEAK                             ⎭
```

4.2.3 External Declarations

```
        external-declaration →
                EXTERNAL external-item , . . . ;

        external-item →
                external-name {: external-attribute . . .}

        external-attribute →
                ⎧ structure-attribute              ⎫
                ⎪ field-attribute                  ⎪
16, 32          ⎪ allocation-unit                  ⎪
                ⎨ extension-attribute              ⎬
32              ⎪ addressing-mode-attribute        ⎪
                ⎪ VOLATILE                         ⎪
32              ⎩ WEAK                             ⎭
```

4.2.4 Forward Declarations

```
        forward-declaration →
                FORWARD forward-item , . . . ;

32, 36  forward-item →
                forward-name {: forward-attribute . . .}

        forward-attribute →
                ⎧ structure-attribute              ⎫
                ⎪ field-attribute                  ⎪
                ⎨ allocation-unit                  ⎬
32              ⎪ extension-attribute              ⎪
                ⎪ addressing-mode-attribute        ⎪
32, 36          ⎩ VOLATILE                         ⎭
```

4.2.5 Local Declarations

```
        local-declaration →
                LOCAL local-item , . . . ;

        local-item →
                local-name {: local-attribute . . .}

        local-attribute →
                ⎧ structure-attribute              ⎫
                ⎪ field-attribute                  ⎪
16, 32          ⎨ allocation-unit                  ⎬
                ⎪ extension-attribute              ⎪
32              ⎪ ALIGN ( boundary-ctce )          ⎪
                ⎩ VOLATILE                         ⎭
```

4.2.6 Stacklocal Declarations

```
stacklocal-declaration →
        STACKLOCAL  local-item , . . . ;
```

4.2.7 Register Declarations

```
register-declaration →
        REGISTER  register-item , . . . ;

register-item →
       { reg-name  { : register-attribute . . . }                          }
       { reg-name  { = ctce }  { : register-attribute . . . }              }

register-attribute →
          ( structure-attribute )
          ( field-attribute     )
16, 32  [ { allocation-unit     }
          ( extension-attribute )
```

4.2.8 Global Register Declarations

```
        [ global-register-declaration →
        [         GLOBAL REGISTER global-reg-item , . . . ;
        [
32, 36  [ global-reg-item →
        [         global-reg-name  =  ctce   { : register-attribute . . . }
        [
        [ register-attribute        — See Section 4.2.7.
```

4.2.9 External Register Declarations

```
        [ external-register-declaration →
        [         EXTERNAL REGISTER external-reg-item , . . . ;
        [
32, 36  [ external-reg-item →
        [         ext-reg-name   { = ctce }  { : register-attribute . . . }
        [
        [ register-attribute     — See Section 4.2.7.
```

4.2.10 Map Declarations

```
map-declaration →
        MAP  map-item , . . . ;

map-item →
        map-name  :  map-attribute . . .

map-attribute →
          ( structure-attribute )
          ( field-attribute     )
16, 32  [ { allocation-unit     }
          ( extension-attribute )
          ( VOLATILE            )
```

4.3 Structure Declarations

```
structure-declaration →
        STRUCTURE structure-definition , . . . ;

structure-definition →
        structure-name
            [ { access-formal , . . . }
              { ; allocation-formal , . . . } ]
            = { [ structure-size-exp ] }
            structure-body

access-formal →
        name

allocation-formal →
        allocation-name   { = allocation-default }

allocation-default →
        ctce

structure-body →
        address-expression   { field-selector }

field-selector    — See Section 2.2.1.
```

4.4 Field Declarations

```
field-declaration →
          FIELD  { field-set-definition }  , . . . ;
                 { field-definition     }

field-set-definition →
          field-set-name  =
               SET
               field-definition , . . .
               TES

field-definition →
          field-name  =  [field-component , . . . ]

field-component →
          ctce
```

4.5 Routine Declarations

```
routine-declaration →
       { ordinary-routine-declaration }
       { global-routine-declaration   }
       { external-routine-declaration }
       { forward-routine-declaration  }
```

4.5.1 Ordinary Routine Declarations

```
ordinary-routine-declaration →
          ROUTINE  routine-definition , . . . ;

routine-definition →
          routine-name  { ( formal-name , . . . ) }
              { : routine-attribute . . . }
              = routine-body

routine-attribute →
       { NOVALUE      }
       { linkage-name }

linkage-name    — See Sections 4.6 and 6.4.

routine-body →
          expression
```

4.5.2 Global Routine Declarations

```
global-routine-declaration →
          GLOBAL ROUTINE
              global-routine-definition , . . . ;

global-routine-definition →
          routine-name  { ( formal-name , . . . ) }
              { : global-routine-attribute . . . }
              = routine-body

global-routine-attribute →
       { NOVALUE      }
       { linkage-name }
32     { WEAK         }

linkage-name    — See Sections 4.6 and 6.4.

routine-body →
          expression
```

4.5.3 External Routine Declarations

```
external-routine-declaration →
          EXTERNAL ROUTINE
              external-routine-item , . . . ;

external-routine-item →
          routine-name  { : ext-routine-attribute . . . }

ext-routine-attribute →
       { NOVALUE                  }
       { linkage-name             }
32   [ { addressing-mode-attribute }
     [ { WEAK                     }

linkage-name    — See Sections 4.6 and 6.4.
```

4.5.4 Forward Routine Declarations

```
forward-routine-declaration →
          FORWARD ROUTINE
              forward-routine-item , . . . ;

forward-routine-item →
          routine-name  { : fwd-routine-attribute . . . }
```

```
      fwd-routine-attribute →
            ⎧NOVALUE                    ⎫
            ⎨linkage-name               ⎬
32          ⎩addressing-mode-attribute  ⎭
```

linkage-name — *See Sections 4.6 and 6.4.*

4.6 Linkage Declarations

```
      linkage-declaration →
            LINKAGE linkage-definition , . . . ;

      linkage-definition →
            linkage-name = linkage-type
                 { ( parameter-location , . . . ) }
                 { : linkage-option . . . }

      linkage-type →
16, 32      ⎧CALL                       ⎫
16     [    ⎪JSR  | EMT  | TRAP         ⎪
       [    ⎨IOT  | INTERRUPT           ⎬
32          ⎪JSB                        ⎪
36          ⎩PUSHJ | F10                ⎭

      parameter-location →
            ⎧STANDARD          ⎫
            ⎨REGISTER = ctce   ⎬
            ⎩nothing           ⎭

      linkage-option →
16          ⎧CLEARSTACK | RTT                        ⎫
32, 36 [    ⎪GLOBAL ( global-segment , . . . )       ⎪
       [    ⎪PRESERVE ( ctce , . . . )               ⎪
       [    ⎨NOPRESERVE ( ctce , . . . )             ⎬
32          ⎪NOTUSED ( ctce , . . . )                ⎪
36     [    ⎪LINKAGE_REGS ( ctce, ctce, ctce )       ⎪
       [    ⎩PORTAL                                  ⎭

32, 36 [  global-segment →
       [        global-register-name = ctce
```

Note: All syntax elements denoted by "ctce" in this section represent register-number expressions. Also, the order in which registers are specified in the LINKAGE_REGS option is: SP, FP, value-return.

4.7 Enable Declarations

```
       [  enable-declaration →
       [        ENABLE handler-name
       [              { ( enable-data-name , . . . ) }
       [
16, 32 [  handler-name       — Must be a declared routine-name.
       [
       [  enable-data-name   — Must be declared in an own-,
       [                       global-, local-, stacklocal-,
       [                       external-, or forward-declaration,
       [                       with the VOLATILE attribute.
```

4.8 Bound Declarations

```
      bound-declaration →
            ⎧literal-declaration           ⎫
            ⎪external-literal-declaration  ⎪
            ⎨bind-data-declaration         ⎬
            ⎩bind-routine-declaration      ⎭
```

4.8.1 Literal Declarations

```
      literal-declaration →
            ⎧LITERAL        ⎫
            ⎩GLOBAL LITERAL ⎭   literal-item , . . . ;

      literal-item →
            literal-name = ctce
                 { : literal-attribute . . . }

      literal-attribute →
            ⎧{SIGNED | UNSIGNED} ( ctce )⎫
32          ⎩WEAK                        ⎭
```

Note: WEAK applies to the GLOBAL form only.

4.8.2 External Literal Declarations

```
      external-literal-declaration →
            EXTERNAL LITERAL
                 external-literal-item , . . . ;

      external-literal-item →
            ext-literal-name { : literal-attribute . . . }
```

```
literal-attribute →
      ⎧ {SIGNED | UNSIGNED} ( ctce ) ⎫
32    ⎩ WEAK                         ⎭
```

4.8.3 Bind Data Declarations

```
bind-data-declaration →
      ⎧ BIND        ⎫
      ⎩ GLOBAL BIND ⎭   bind-data-item , . . . ;

bind-data-item →
          bind-data-name = expression
             { : bind-data-attribute . . . }

bind-data-attribute →
          ⎧ structure-attribute ⎫
          ⎪ field-attribute     ⎪
16, 32 [  ⎨ allocation-unit     ⎬
       [  ⎪ extension-attribute ⎪
          ⎪ VOLATILE            ⎪
32        ⎩ WEAK                ⎭
```

Note: WEAK applies to the GLOBAL form only.

4.8.4 Bind Routine Declarations

```
bind-routine-declaration →
      ⎧ BIND ROUTINE        ⎫
      ⎩ GLOBAL BIND ROUTINE ⎭
             bind-routine-item , . . . ,

bind-routine-item →
          bind-routine-name = expression
             { : bind-routine-attribute . . . }

bind-routine-attribute →
          ⎧ NOVALUE      ⎫
          ⎨ linkage-name ⎬
32        ⎩ WEAK         ⎭
```

Note: WEAK applies to the GLOBAL form only.

linkage-name — *See Sections 4.6 and 6.4.*

4.9 Compiletime Declarations

```
compiletime-declaration →
          COMPILETIME compiletime-item , . . . ;

compiletime-item →
          compiletime-name = initial-value

initial-value →
          ctce
```

Note: A compiletime-name value may be changed during compilation by the %ASSIGN lexical function (see Section 5.3).

4.10 Macro Declarations

```
macro-declaration →
      ⎧ keyword-macro-declaration    ⎫
      ⎩ positional-macro-declaration ⎭
```

4.10.1 Keyword Macro Declarations

```
keyword-macro-declaration →
          KEYWORDMACRO keyword-macro-definition , . . .

keyword-macro-definition →
          keyword-macro-name ( keyword-formal , . . . )
             = macro-body %

keyword-formal →
          name { = keyword-default-parameter }

keyword-default-parameter →
          { lexeme . . . }

macro-body →
          any-lexeme-except-% . . .
```

4.10.2 Positional Macro Declarations

```
positional-macro-declaration →
        MACRO positional-macro-definition , . . . ;

positional-macro-definition →
      ⎧ simple-macro      ⎫
      ⎨ conditional-macro ⎬
      ⎩ iterative-macro   ⎭

simple-macro →
        macro-name  { ( name , . . . ) }
            = macro-body %

conditional-macro →
        macro-name  { ( name , . . . ) }  [ ]
            = macro-body %

iterative-macro →
        macro-name  { ( name , . . . ) }
            [ name , . . . ]
            = macro-body %

macro-body →
        any-lexeme-except-% . . .
```

4.11 Require Declarations

```
require-declaration →
        REQUIRE file-designator;

file-designator →
        quoted-string
```

4.12 Library Declarations

```
library-declaration →
        LIBRARY file-designator;

file-designator →
        quoted-string
```

4.13 Psect Declarations

```
psect-declaration →
        PSECT psect-definition , . . . ;

psect-definition →
        storage-class = psect-name
            { ( psect-attribute , . . . ) }

storage-class →
      ⎧ OWN    ⎫
16, 32⎨ GLOBAL ⎬
      ⎪ PLIT   ⎪
      ⎩ CODE   ⎭

psect-attribute →
      ⎧ EXECUTE | NOEXECUTE        ⎫
      ⎪ WRITE | NOWRITE            ⎪
      ⎪ OVERLAY | CONCATENATE      ⎪
      ⎪ LOCAL | GLOBAL             ⎪
      ⎨ READ | NOREAD              ⎬
32    ⎪ SHARE | NOSHARE            ⎪
      ⎪ PIC | NOPIC                ⎪
      ⎪ ALIGN ( boundary-ctce )    ⎪
      ⎩ addressing-mode-attribute  ⎭
```

4.14 Switches Declarations

```
switches-declaration →
                   ⎧ on-off switch-item    ⎫
        SWITCHES   ⎨                       ⎬   . . . ;
                   ⎩ special-switch-item   ⎭

on-off-switch-item →
      ⎧ ERRS | NOERRS            ⎫
      ⎪ OPTIMIZE | NOOPTIMIZE    ⎪
      ⎨ SAFE | NOSAFE            ⎬
      ⎪ ZIP | NOZIP              ⎪
      ⎩ UNAMES | NOUNAMES        ⎭

special-switch-item →
      ⎧ LANGUAGE ( language-list )              ⎫
      ⎨ LINKAGE ( linkage-name )                ⎬
      ⎪ LIST ( list-option , . . . )            ⎪
32    ⎩ ADDRESSING_MODE ( mode-spec , . . . )   ⎭
```

language-list — *See Section 1.1.*

linkage-name — *See Sections 4.6 and 6.4.*

list-option — *See Section 1.1.*

mode-spec — *See Section 1.1.*

4.15 Label Declarations

```
label-declaration →
        LABEL label-name , . . . ;
```

4.16 Builtin Declarations

```
builtin-declaration →
        BUILTIN builtin-name , . . . ;
```

builtin-name — *See Section 7 for a summary of the machine-specific names that may be declared as builtin-names.*

4.17 Undeclare Declarations

```
undeclare-declaration →
        UNDECLARE undeclared-name , . . . ;
```

5.0 LEXICAL PROCESSING FACILITIES

The compile-time features described in this section allow conditional compilation of alternative portions of the source text, and allow extensive modification and expansion of the source text during the compilation process. These features are lexical conditionals, lexical functions, and macro calls (in conjunction with the macro-declaration facility).

5.1 Lexical Conditionals

```
lexical-conditional →
        %IF lexical-test
            %THEN consequent-lexeme . . .
          { %ELSE alternative-lexeme . . . }
          { nothing                        }
            %FI
```

```
lexical-test →
        ctce
```

Note: Either the consequent-lexeme or the alternative-lexeme may be null.

5.2 Lexical Functions

The thirteen categories of lexical-functions are as follows:

String Functions
Delimiter Functions
Name Functions
Sequence-Test Functions
Bits Functions
Allocation Functions
Fieldexpand Functions
Calculation Functions
Compiler-State Functions
Advisory Functions
Title Functions
Quote Functions
Macro Functions

The individual functions corresponding to these categories are given below, with brief semantic descriptions of each.

String Functions

%CHAR (ctce , . . .)

Returns a quoted-string formed by interpreting the numeric value of each ctce as a single ASCII character code, and concatenating the corresponding characters. E.g., %CHAR(65,66,67,39,97,98,99) is replaced by 'ABC''abc'.

%STRING (string-param , . . .)

Returns a single quoted-string formed by concatenating the characters represented by each string-param. Each string-param, after evaluation, must result in a quoted-string, a name, a numeric-literal, or a null lexeme. E.g., %STRING(23,%B'-111') is replaced by '23-7'.

%EXACTSTRING (length, fill, string-param , . . .)

Returns a quoted-string as formed by %STRING, but either truncated or extended on the right as specified by the 'length' ctce value, and filled if necessary as specified by the 'fill' ctce value (interpreted as for %CHAR). E.g., %EXACTSTRING(6,%C'9','ABC') is replaced by 'ABC999'.

%CHARCOUNT (string-param , . . .)

Evaluates string-params as for %STRING, and returns a numeric-literal equal to the count of characters within the resulting string. E.g., %CHARCOUNT('A''C',23) is replaced by 5.

Delimiter Functions

%EXPLODE (string-param , . . .)

Forms an intermediate quoted-string from string-params as for %STRING, and returns a comma-separated list of quoted-strings, each consisting of a single character of the intermediate string (in corresponding sequence). E.g., %EXPLODE('ABC',%O'77') is replaced by 'A','B', 'C','6','3'; i.e., 9 lexemes.

%REMOVE (parameter)

Returns the indicated parameter after removing any enclosing (and matched) parentheses, square brackets ([ ]), or angle brackets (<>).

Name Functions

%NAME (string-param , . . .)

Returns a name formed by the characters represented by the string-params, which are interpreted as for %STRING. E.g., %NAME(' 302',beta) is replaced by 302BETA (as a name).

Sequence-Test Functions

%NULL (parameter , . . .)

Returns the literal 1 if all of the given parameters are null; returns 0 otherwise. E.g., %NULL(ALPHA,,DELTA) is replaced by 0.

%IDENTICAL (parameter , parameter)

Returns the literal 1 if the two parameters (after evaluation as for a normal macro call) consist of identical lexeme sequences; returns 0 otherwise. E.g., %IDENTICAL(A+B,a+b) is replaced by 1.

Bits Functions

%NBITS (ctce , . . .)

Returns the minimum number of bits needed to represent any of the ctce parameters (i.e., including the largest), interpreted as signed integers, in a sign-extended field. E.g., %NBITS(7,2) is replaced by 3. %NBITS(-8) is replaced by 4.

%NBITSU (ctce , . . .)

Returns the minimum number of bits needed to represent any of the ctce parameters (i.e., including the largest), interpreted as unsigned integers, in a zero-extended field. E.g., %NBITSU(7,2) is replaced by 3. %NBITSU (-8,7) is replaced by %BPVAL.

Allocation Functions

%ALLOCATION (data-segment-name)

Returns the number of storage units allocated for the specified data segment. E.g., using the BLISS-32 compiler: %ALLOCATION(X) with X declared LONG is replaced by 4.

%SIZE (structure-attribute)

Returns the number of storage units that would be allocated for a data structure declared with the specified structure-attribute: E.g., using the BLISS-32 compiler: %SIZE(VECTOR[10,WORD]) is replaced by 20.

Fieldexpand Functions

%FIELDEXPAND (field-name {, ctce})

Returns the *nth* field-component value of field-name, where n is specified by ctce as n−1. If the ctce parameter is null, all field-component values of the field-name are returned, as a comma-separated list. E.g., If DCB_C=[0,11,16,3] , then %FIELDEXPAND(DCB_C,2) would be replaced by 16.

Calculation Functions

%ASSIGN (compiletime-name , ctce)

Assigns the ctce value as the new value of the specified COMPILETIME name.

%NUMBER (number-param)

Returns a numeric-literal formed from the value represented by number-param, which must be either a numeric-literal, a literal-name, or a quoted-string consisting of decimal digits with optional sign. E.g., %NUMBER(%O'100') is replaced by 64.

Compiler-State Functions

%DECLARED (name)

Returns the literal 1 if the given name lexeme is a user-declared name (i.e., not predeclared); returns 0 otherwise.

%SWITCHES (on-off-switch-name , . . .)

Returns the literal 1 if all given on-off-switch-names match the current on-off-switch settings; returns 0 otherwise.

%BLISS (language-name)

Returns the literal 1 if the given language-name corresponds to the compiler processing the module; returns 0 otherwise. (Valid language-names are BLISS16, BLISS32, and BLISS36.)

%VARIANT

Returns a numeric-literal representing the setting of /VARIANT in the compilation command.

Advisory Functions

%ERROR (string-param , . . .)

Causes an error diagnostic to be produced (by the compiler) from the string-params, processed as by %STRING.

%ERRORMACRO (string-param , . . .)

Causes an error diagnostic to be produced (by the compiler) from the string-params, processed as by %STRING, and causes all currently active macro expansions to be terminated.

%WARN (string-param , . . .)

Causes a warning diagnostic to be produced (by the compiler) from the string-params, processed as by %STRING.

%INFORM (string-param , . . .)

Causes an informational diagnostic to be produced (by the compiler) from the string-params, processed as by %STRING.

%PRINT (string-param , . . .)

Causes a line to be included in the listing file (if any) consisting of the string-params, processed as by %STRING.

Title Functions

%TITLE quoted-string

Incorporates the quoted-string into the title portion of listing-page header.

%SBTTL quoted-string

Incorporates the quoted-string into the subtitle portion of listing-page header.

Quote Functions

%QUOTE

Inhibits lexical binding of the lexeme following the function name.

%UNQUOTE

Forces lexical binding of the lexeme following the function name, even where it would not normally be bound.

%EXPAND

Forces lexical binding of the lexeme following the function name; and, if that lexeme is itself a macro or lexical-function name, expands the macro call or evaluates the function.

Macro Functions

%REMAINING

Returns a comma-separated list consisting of any actual parameters of the call that, during expansion, are not yet associated with formal parameters.

%LENGTH

Returns the number of actual parameters in the call.

%COUNT

Returns the recursion depth if within a conditional macro, or the number of completed iterations if within an iterative macro.

%EXITITERATION

Terminates expansion of the current iteration of an iterative macro call. (For a noniterative expansion, this function is equivalent to %EXITMACRO.)

%EXITMACRO

Terminates expansion of a macro call.

5.3 Macro Calls

```
macro-call →
    { keyword-macro-call    }
    { positional-macro-call }
```

5.3.1 Keyword Macro Calls

```
keyword-macro-call →
        name
          { (  keyword-macro-actual , . . . ) }
          { [  keyword-macro-actual , . . . ] }
          { <  keyword-macro-actual , . . . > }

keyword-macro-actual →
        keyword-formal-name =  {lexeme . . .}
```

5.3.2 Positional Macro Calls

```
positional-macro-call →
        name
          { nothing          }
          { (  lexeme . . . ) }
          { [  lexeme . . . ] }
          { <  lexeme . . . > }
```

6.0 PREDECLARED NAMES

6.1 Literals

The literal names and values predeclared in every module are:

Name	Value		
	BLISS-16	BLISS-32	BLISS-36
%BPVAL	16	32	36
%BPADDR	16	32	18
%BPUNIT	8	8	36
%UPVAL	2	4	1

6.2 Macros

The macros predeclared in every module are:

For BLISS-16:

```
%BLISS16 [] = %REMAINING %
%BLISS32 [] = %
%BLISS36 [] = %
```

For BLISS-32:

```
%BLISS16 [] = %
%BLISS32 [] = %REMAINING %
%BLISS36 [] = %
```

For BLISS-36:

```
%BLISS16 [] = %
%BLISS32 [] = %
%BLISS36 [] = %REMAINING %
```

6.3 Structures

The structures predeclared in every module are:

For BLISS-16:

```
STRUCTURE
        VECTOR[I;N,UNIT=2,EXT=0] =
            [N*UNIT]
            (VECTOR+I*UNIT)<0,8*UNIT,EXT>,

        BLOCK[O,P,S,E;BS,UNIT=2] =
            [BS*UNIT]
            (BLOCK+O*UNIT)<P,S,E>,

        BLOCKVECTOR[I,O,P,S,E;N,BS,UNIT=2] =
            [N*BS*UNIT]
            (BLOCKVECTOR+(I*BS+O)*UNIT)<P,S,E>,

        BITVECTOR[I;N] =
            [((N+15)/16)*2]
            (BITVECTOR+I/16)<I MOD 16,1,0>;
```

For BLISS-32:

```
STRUCTURE
        VECTOR[I;N,UNIT=4,EXT=0] =
            [N*UNIT]
            (VECTOR+I*UNIT)<0,8*UNIT,EXT>,

        BLOCK[O,P,S,E;BS,UNIT=4] =
            [BS*UNIT]
            (BLOCK+O*UNIT)<P,S,E>,

        BLOCKVECTOR[I,O,P,S,E;N,BS,UNIT=4] =
            [N*BS*UNIT]
            (BLOCKVECTOR+(I*BS+O)*UNIT)<P,S,E>,

        BITVECTOR[I;N] =
            [(N+7)/8]
            (BITVECTOR)<I,1,0>;
```

For BLISS-36:

```
STRUCTURE
        VECTOR[I;N] =
            [N]
            (VECTOR+I)<0,36>,

        BLOCK[O,P,S,E;BS] =
            [BS]
            (BLOCK+O)<P,S,E>,

        BLOCKVECTOR[I,O,P,S,E;N,BS] =
            [N*BS]
            (BLOCKVECTOR+O+I*BS)<P,S,E>,

        BITVECTOR[I;N] =
            [(N+35)/36]
            (BITVECTOR+I/36)<I MOD 36, 1, 0>;
```

6.4 Linkages and Linkage-Functions

The predeclared linkage-names are:

BLISS-16	BLISS-32	BLISS-36
BLISS FORTRAN FORTRAN_FUNC FORTRAN_SUB	BLISS FORTRAN FORTRAN_FUNC FORTRAN_SUB	BLISS36C BLISS10 FORTRAN_FUNC FORTRAN_SUB

The default linkage-name for BLISS-16 and BLISS-32 is BLISS; for BLISS-36 the default linkage-name is BLISS36C.

The following linkage-functions are predefined and can be declared in a BUILTIN declaration for use in routines that have the CALL or F10 linkage-type:

	ACTUALCOUNT( )	—	No. of actual params. in call
	ACTUALPARAMETER(i)	—	Value of i*th* parameter
	ARGPTR( )	—	Address of argument block
16, 32	NULLPARAMETER(i)	—	1 if i*th* param. is null; 0 otherwise

6.5 Supplementary Functions

The supplementary character-handling functions predeclared in every module are ("CS" stands for "character sequence"):

CH$PTR (addr, i, chsize)
— *Create a CS-pointer*

CH$PLUS (ptr, i)
— *Increment a CS-pointer*

CH$DIFF (ptr1, ptr2)
— *Take difference of two CS-pointers*

CH$RCHAR (ptr)
— *Fetch a character*

CH$WCHAR (char, ptr)
— *Assign a character*

CH$RCHAR_A (addr)
— *Fetch a character, then advance CS-pointer*

CH$WCHAR_A (char, addr)
— *Assign a character, then advance CS-pointer*

CH$A_RCHAR (addr)
— *Advance CS-pointer, then fetch a character*

CH$A_WCHAR (char, addr)
— *Advance CS-pointer, then assign a character*

CH$ALLOCATION (n, chsize)
— *Storage allocation for given number of characters*

CH$SIZE (ptr)
— *Number of bits per character (i.e., returns character size)*

CH$MOVE (n, sptr, dptr)
— *Move a character sequence*

CH$COPY (sn1, sptr1, sn2, sptr2 , . . . , fill, dn, dptr)
— *Move and concatenate a series of character sequences*

CH$FILL (fill, dn, dptr)
— *Initialize character sequence with fill character*

CH$LSS (n1, ptr1, n2, ptr2, fill)
— *Compare character sequences for less than*

CH$LEQ (n1, ptr1, n2, ptr2, fill)
— *Compare character sequences for less than or equal*

CH$GTR (n1, ptr1, n2, ptr2, fill)
— *Compare character sequences for greater than*

CH$GEQ (n1, ptr1, n2, ptr2, fill)
— *Compare character sequences for greater than or equal*

CH$EQL (n1, ptr1, n2, ptr2, fill)
— *Compare character sequences for equal*

CH$NEQ (n1, ptr1, n2, ptr2, fill)
— *Compare character sequences for not equal*

CH$COMPARE (n1, ptr1, n2, ptr2, fill)
— *Compare character sequences for less than, equal to, or greater than. (The value returned is −1, 0, or 1 respectively.)*

CH$FIND_SUB (cn, cptr, pn, pptr)
— *Find given sub-sequence*

CH$FIND_CH (n, ptr, char)
— *Find given character*

CH$FIND_NOT_CH (n, ptr, char)
— *Find first character other than given character*

CH$TRANSTABLE (trans-string)
— *Create translation table*

CH$TRANSLATE (tab, sn, sptr, fill, dn, dptr)
— *Translate using translation table*

CH$FAIL (ptr)
— *Test for failure to satisfy search*

7.0 MACHINE SPECIFIC NAMES

The following names may be declared by means of the BUILTIN declaration:

```
builtin-name →
      ⎧register-name              ⎫
      ⎨machine-specific-function  ⎬
      ⎩linkage-function           ⎭
```

For BLISS-16:

```
     ⎡ register-name →
     ⎢     {R0 | R1 | --- | R5 | SP | PC}
     ⎢
     ⎢ machine-specific-function →
     ⎢     ⎧DECX                 ⎫
     ⎢     ⎪HALT                 ⎪
     ⎢     ⎪MFPD | MFPI | MFPS   ⎪
16   ⎢     ⎨MTPD | MTPI | MTPS   ⎬
     ⎢     ⎪RESET | ROT          ⎪
     ⎢     ⎪SWAB                 ⎪
     ⎢     ⎩WAIT                 ⎭
     ⎢
     ⎣ linkage-function   — See Section 6.4.
```

For BLISS-32:

```
     ⎡ register-name →
     ⎢     {R0 | R1 | --- | R11 | AP | FP | SP | PC}
     ⎢
     ⎢ machine-specific-function →
     ⎢     ⎧ADAWI | ASHQ                               ⎫
     ⎢     ⎪BICPSW | BISPSW | BPT                      ⎪
     ⎢     ⎪CALLG | CHME | CHMK | CHMS                 ⎪
     ⎢     ⎪CHMU | CMPD | CMPF | CMPP                  ⎪
     ⎢     ⎪CRC | CVTDF | CVTDL | CVTFD                ⎪
     ⎢     ⎪CVTFL | CVTLD | CVTLF | CVTLP              ⎪
     ⎢     ⎪CVTPL | CVTPT | CVTRDL | CVTRFL            ⎪
     ⎢     ⎪CVTPS | CVTSP | CVTTP                      ⎪
32   ⎢     ⎨EDITPC | EDIV | EMUL                       ⎬
     ⎢     ⎪FFC | FFS | HALT                           ⎪
     ⎢     ⎪INDEX | INSQUE                             ⎪
     ⎢     ⎪MFPR | MOVP | MOVPSL | MOVTUC              ⎪
     ⎢     ⎪MTPR | NOP | PROBER | PROBEW               ⎪
     ⎢     ⎪REMQUE | ROT | SCANC | SPANC               ⎪
     ⎢     ⎪TESTBITCC | TESTBITCCI | TESTBITCS         ⎪
     ⎢     ⎩TESTBITSC | TESTBITSS | TESTBITSSI         ⎭
     ⎢
     ⎣ linkage-function   — See Section 6.4.
```

7.0 MACHINE SPECIFIC NAMES, Continued

For BLISS-36:

```
     ⎡ register-name →
     ⎢     {AP | FP | SP}
     ⎢
     ⎢ machine-specific-function →
     ⎢     ⎧ASH                                        ⎫
     ⎢     ⎪COPYII | COPYIN | COPYNI | COPYNN          ⎪
     ⎢     ⎪DPB                                        ⎪
36   ⎢     ⎪FIRSTONE                                   ⎪
     ⎢     ⎨INCP                                       ⎬
     ⎢     ⎪LDB | LSH                                  ⎪
     ⎢     ⎪MACHOP | MACHSKIP                          ⎪
     ⎢     ⎪POINT                                      ⎪
     ⎢     ⎪REPLACEI | REPLACEN | ROT                  ⎪
     ⎣     ⎩SCANI | SCANN                              ⎭

       linkage-function   — See Section 6.4.
```

8.0 NAMES RESERVED FOR SPECIAL PURPOSES

The following names are reserved for future extensions.

For all dialects:

BIT
IOPAGE
PRESET
RECORD
SHOW

Additional for BLISS-16:

ALIGN
LONG
WEAK

Additional for BLISS-36:

ADDRESSING_MODE
ALIGN
BYTE
ENABLE
LONG
PSECT
WEAK
WORD

COMMAND SUMMARY

1.0 BLISS-32 COMMANDS

This section describes the VAX/VMS command for invoking a BLISS-32 compilation. The notational conventions used earlier in this guide are used in this section, with the following additional convention:

- The symbol "+ . . ." denotes an optional repetition of the immediately preceding item (always a source file-spec), with successive instances separated by "+".

1.1 Command Line Syntax

The syntax of the BLISS-32 compilation-request command, given following a command-level prompt ($) in interactive mode, is:

```
compilation-request ⟶
        BLISS {qualifier . . .} ƀ input-spec , . . .

ƀ ⟶
        {blank | tab} . . .

input-spec ⟶
        file-spec + . . . {qualifier}

qualifier ⟶
      ⎧ output-qualifier              ⎫
      ⎪ general-qualifier             ⎪
      ⎨ terminal-qualifier            ⎬
      ⎪ optimization-qualifier        ⎪
      ⎪ source-list-qualifier         ⎪
      ⎩ machine-code-list-qualifier   ⎭
```

Note: The individual qualifiers are described in Section 1.2.

Usage Rules:

1. Each input-spec given in the command implies a separate compilation. That is, unless the /NOCODE qualifier is specified, one object- or library-module file is produced for each input-spec.

2. If no qualifiers (or corresponding module-head switches) are specified, the default compilation results are as follows:

   - One or more object-module files (one per input-spec). Each object-module file takes its name from the corresponding input file name, and the default file-type OBJ is appended.
   - No listing file is produced.
   - Error messages, if any, and a compilation summary are reported at the user's terminal.

3. If an input-spec consists of two or more file-specs separated by plus signs, the specified files are concatenated and processed as one source module. That is, the specified files are assumed to contain, collectively, an entire source module. The object-module file in this case takes its name from the first source file name specified in the input-spec.

4. If the file type is omitted in an input-file specification, the default type B32 is assumed first and then, if necessary, type BLI. (See Sections 1.2 and 1.3 for type defaults for a library precompilation.)

5. One or more blanks or tabs may be used anywhere that a space appears in the command-line syntax definitions. (The only mandatory space is indicated by the symbol "ƀ" in the syntax rule.)

1.2 Qualifiers

The default qualifiers and values for interactive-mode compilations are underlined.

Output Qualifiers

```
output-qualifier ⟶
   ⎧ /OBJECT {= file-spec} | /NOOBJECT       ⎫
   ⎨ /LIST {= file-spec} | /NOLIST           ⎬
   ⎩ /LIBRARY {= file-spec} | /NOLIBRARY     ⎭
```

Usage Rules:

1. The /OBJECT and /LIBRARY qualifiers are mutually exclusive, i.e., only one of the two may be specified.

2. If the /LIBRARY qualifier is specified, a library-file precompilation is performed (as opposed to an object-module compilation).

3. If an output-file type is not specified, the default types are OBJ, LIS, and L32, for the obiect, listing, and library file respectively.

4. If a file-spec is not given in an output qualifier, the output file takes the name of the corresponding input file, with the appropriate default type (Rule 3).

General Qualifiers

```
general-qualifier →
  ⎧ /CODE | /NOCODE                ⎫
  ⎨ /DEBUG | /NODEBUG              ⎬
  ⎪ /TRACEBACK | /NOTRACEBACK      ⎪
  ⎩ /VARIANT {= n}                 ⎭
```

Usage Rules:

1. /NOCODE implies a syntax check only.

2. /DEBUG implies full symbol-table information for the symbolic debugger.

3. /NOTRACEBACK implies no symbol-table information for the debugger, and nullifies the effect of /DEBUG, if specified. It produces the most compact object module and is appropriate for final production compilations.

4. If /VARIANT is not specified, a %VARIANT value of 0 is assumed. If /VARIANT is specified without a value, a %VARIANT value of 1 is implied. If a value (n) is specified, it must be a decimal integer within the value range of a signed longword.

Terminal Qualifier

```
terminal-qualifier →
                      ⎧ terminal-value              ⎫
         /TERMINAL =  ⎨                             ⎬
                      ⎩ ( terminal-value , . . . )  ⎭

terminal-value →
     ⎧ ERRORS | NOERRORS               ⎫
     ⎩ STATISTICS | NOSTATISTICS       ⎭
```

Usage Rules:

1. If /TERMINAL is not specified, the underlined defaults for terminal-value are assumed.

2. If NOERRORS is specified, compilation errors are not reported at the user's terminal.

3. If STATISTICS is specified, the name and size of each routine is reported as it is compiled.

Optimize Qualifier

```
optimize-qualifier →
                      ⎧ optimize-value              ⎫
         /OPTIMIZE =  ⎨                             ⎬
                      ⎩ ( optimize-value , . . . )  ⎭

optimize-value →
     ⎧ LEVEL: {0 | 1 | 2 | 3}     ⎫
     ⎨ QUICK | NOQUICK            ⎬
     ⎪ SAFE | NOSAFE              ⎪
     ⎩ SPACE | SPEED              ⎭
```

Usage Rules:

1. The several optimize-value alternatives affect the compiler's optimization strategies; see the *BLISS-32 User's Guide* for full details.

2. The optimize-values SPEED and SPACE are mutually exclusive.

3. QUICK implies the omission of some standard optimizations in favor of increased compilation speed.

4. SAFE (a default) implies that all named data-segments are referenced by name only, i.e., not by computed addresses. If this is not true for a given module, the NOSAFE alternative should be specified.

Source List Qualifier

```
source-list-qualifier →
                         ⎧ source-value              ⎫
         /SOURCE_LIST =  ⎨                           ⎬
                         ⎩ ( source-value , . . . )  ⎭

source-value →
     ⎧ HEADER | NOHEADER                           ⎫
     ⎪ PAGE_SIZE: {20 | 21 | . . . | 52}           ⎪
     ⎪ LIBRARY | NOLIBRARY                         ⎪
     ⎨ REQUIRE | NOREQUIRE                         ⎬
     ⎪ EXPAND_MACROS | NOEXPAND_MACROS             ⎪
     ⎪ TRACE_MACROS | NOTRACE_MACROS               ⎪
     ⎩ SOURCE | NOSOURCE                           ⎭
```

Usage Rules:

1. The several source-value alternatives affect the form and content of the source portion of the output listing; see the *BLISS-32 User's Guide* for full details.

2. The /LIST output-qualifier (or the corresponding module-head switch) must be in effect in order for any source-value alternatives to be meaningful.

Machine-Code List Qualifier

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

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

Usage Rules:

1. The several code-value alternatives affect the form and content of the object portion of the output listing; see the *BLISS-32 User's Guide* for full details.

2. The /LIST output-qualifier (or the corresponding module-head switch) must be in effect in order for any code-value alternatives to be meaningful.

3. NOOBJECT is mutually exclusive with any other code-value alternative.

1.3 Summary of File Type Defaults

The ordered lists of successive type defaults assumed by the compiler for various input files, and the type defaults applied by the compiler for output files, are given below.

Input-File Type Defaults

Source File For:	Default Type List
● Object-module compilation	.B32, .BLI
● Library-file precompilation	.R32, .REQ, .B32, .BLI
File Specified In:	
● REQUIRE declarations	.R32, .REQ, .B32, .BLI
● LIBRARY declarations	.L32

Output-File Type Defaults

● Object-module file	.OBJ
● Library file	.L32
● Listing file	.LIS

2.0 BLISS-16C AND BLISS-36 COMMANDS

The following monitor-level commands invoke the BLISS-16C and BLISS-36 compilers on a DECsystem-10 with appropriate BLISS support :

```
.R BLS16C    –    for BLISS-16C

.R BLISS     –    for BLISS-36
```

(The "." represents the monitor-mode prompt character.) The compiler responds with a "*" prompt, requesting a compilation command line.

2.1 Command Line Syntax

The syntax of a direct-compilation-request command line is:

```
direct-compilation-request →
      { output-file-list }
            = source-file-list { switch-item . . . }

output-file-list →
      ⎧ object-filespec,listing-filespec  ⎫
      ⎪ library-filespec,listing-filespec ⎪
      ⎨ object-filespec                   ⎬
      ⎪ library-filespec                  ⎪
      ⎩ ,listing-filespec                 ⎭

source-file-list →
      source-filespec , . . .
```

Note: The switch-items applicable to each compiler are described separately below.

Usage Rules:

1. If the /LIBRARY switch is specified in the command line, then a library-file specification is applicable in the output-file list (and a library-file extension default will be assumed, if one is required). Otherwise, an object-file specification is applicable.

2. If either one or both of the output filespecs (object/library or listing) are omitted, the corresponding output file(s) are not produced.

2.0 BLISS-16C AND BLISS-36 COMMANDS, Continued

3. If more than one source file is specified, these files will be logically concatenated by the compiler and treated as one source file. Program modules need not be terminated at file boundaries, and may consist of more than one source file.

4. If incomplete file specifications are given, standard TOPS-10 defaults will be applied to the missing portions (such as device name, project-programmer number, and/or protection code) as appropriate.

5. If file extensions are omitted for one or more files, default extensions will be assumed according to the rules given below for each compiler.

6. Default switch settings are assumed as described below for each compiler.

Exception for BLISS-16C: If only one output-file is specified (in either the object or listing position) and /LIBRARY is not specified, the file is taken to be a file for subsequent assembler input and will have the default extension .P11.

The syntax of an indirect-compilation-request command line is:

```
@ indirect-filespec
```

where indirect-filespec specifies a file containing one or more direct or indirect command lines. (In the case of BLISS-16C, only the direct compilation request encountered first is processed.)

File Extension Defaults for BLISS-16C

File	Default Extensions
Source file for object compilation	.B16, .BLI, then null
Source file for library compilation	.R16, .REQ, .B16, BLI, then null
Object file	.OBJ
Library file	.L16
Listing file	.LST
Indirect file	.CMD

File Extension Defaults for BLISS-36

File	Default Extensions
Source file for object compilation	.B36, then .BLI
Source file for library compilation	.R36, .REQ, .B36, then .BLI
Object file	.REL
Library file	.L36
Listing file	.LST
Indirect file	.CMD

2.2 Command Switches

The command switches are presented below in the format used previously for language syntax. Default switch settings, where applicable, are underlined. The three categories of command switches are the on-off-switches, the special-switches (which are either the same as or very similar to the corresponding module-switches; see Section 1.1), and the command-line-only switches. Switches in any of these categories can be given in any order.

On-Off-Switches

```
on-off-switch →
    { /CODE | /NOCODE           }
    { /DEBUG | /NODEBUG         }
    { /ERRS | /NOERRS           }
    { /OPTIMIZE | /NOOPTIMIZE   }
    { /SAFE | /NOSAFE           }
    { /UNAMES | /NOUNAMES       }
    { /ZIP | /NOZIP             }
```

Special-Switches

```
special-switch →
       { /LIST: {list-option | (list-option , . . . )} }
16 [   { /ADDRESS: mode-16-option                      }
   [   { /OBJECT: object-option                        }
       { /OPTLEVEL: opt-level-option                   }
```

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

16 [ mode-16-option →
   [      {ABSOLUTE | RELATIVE}

16 [ object-option →
   [      {ABSOLUTE | RELOCATABLE}
```

Command-Line-Only Switches

```
command-line-only-switch →
       { /LIBRARY | /NOLIBRARY               }
       { /PAGESIZ:n                          }
       { /STATISTICS | /NOSTATISTICS         }
       { /VARIANT:n                          }
16 [   { /BLISS11:"bliss-11-switches"        }
   [   { /COMPRESS | /NOCOMPRESS             }
36     { /HEADER | /NOHEADER                 }
16 [   { /IDELETE | /NOIDELETE               }
   [   { /INTER:intermediate-filespec        }
36 [   { KA10 | KI10 | KL10                  }
   [   { /QUICK | /NOQUICK                   }
16     { /RBLISS11 | /NORBLISS11             }
36     { TOPS10 | TOPS20                     }
```

2.0 BLISS-16C AND BLISS-36 COMMANDS, Continued

Note: (1) The range of /PAGESIZ:n is 20 through 52, inclusive; the default value is 52.

*(2) If no /VARIANT switch is given, the value of %VARIANT is set to 0. If /VARIANT is given without an ":n" argument, the value of %VARIANT is set to 1. If n is specified, it must be a decimal integer within the range of a BLISS value for the compiler in question, that is, −(2**%BPVAL−1) ≤ n ≤ (2**%BPVAL−1)−1.*