VSI OpenVMS BLISS Language Reference Manual

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

VSI OpenVMS BLISS Language Reference Manual

VMS Software

Legal Notice

Table of Contents

Preface

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

2. Intended Audience

This manual is primarily intended for system programmers, including those whose programming tasks would traditionally imply the use of assembly language. It is also addressed to other programmers for whom the transportability of programs between several BLISS target systems is of prime concern. Familiarity with the basic architecture of one or more of the target systems is assumed; however, familiarity with the relevant assembly language is not assumed.

3. Related Documents

The following documents provide additional information relating to the linking, execution, and debugging of BLISS–32 programs under OpenVMS operating systems:

- *VSI OpenVMS Linker Utility Manual* [https://docs.vmssoftware.com/vsi-openvms-linker-utility-manual/]
- *VSI OpenVMS DCL Dictionary: A–M* [https://docs.vmssoftware.com/vsi-openvms-dcl-dictionary-a-m/] and *VSI OpenVMS DCL Dictionary: N–Z* [https://docs.vmssoftware.com/vsi-openvms-dcl-dictionary-n-z/]
- *VSI OpenVMS Debugger Manual* [https://docs.vmssoftware.com/vsi-openvms-debugger-manual/]

4. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

5. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

6. Conventions

The following conventions are used in this manual:

Convention	Meaning
Ctrl/*x*	A sequence such as **Ctrl/*x*** indicates that you must hold down the key labeled **Ctrl** while you press another key or a pointing device button.

Convention	Meaning
Return	A key name enclosed in a box indicates that you press a key on the keyboard.
…	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
⋮	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
()	In command format descriptions, parentheses indicate that you must enclose the options in parentheses if you choose more than one.
[]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
|	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are options; within braces, at least one choice is required. Do not type the vertical bars on the command line.
Bold Type	Bold type represents the name of an argument, an attribute, or a reason. It also represents the introduction of a new term.
Italic Type	Italic type indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (**/PRODUCER= *name***), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TYPE	Uppercase type indicates the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Monospace Type`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
`Bold Monospace Type`	Bold monospace type indicates a command, command qualifier, or statement.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
Numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes – binary, octal, or hexadecimal – are explicitly indicated.

Chapter 1. Introduction

BLISS is a system implementation language for three DIGITAL computer families:

- PDP–11 (16-bit)
- VAX (32-bit)
- DECsystem–10 and DECSYSTEM–20 (36-bit)

Because of the dissimilarities among these target systems, BLISS has three dialects: BLISS–16, BLISS–32, and BLISS–36. The numeric suffix indicates the word length, in bits, of the respective target system.

BLISS is classified as a system implementation language (rather than an application-oriented language) because BLISS is primarily intended for building system software, such as operating systems, compilers, utilities, and real-time processors. Such software is often large and complicated, is often close to the hardware, and is usually very sensitive to efficiency. In addition, most system software is very frequently used by many individuals (in some cases with an unpredictable variety of input data), and therefore must be highly dependable.

This chapter discusses BLISS concepts and introduces BLISS language features.

1.1. BLISS Dialects

Each BLISS dialect is supported by a separate compiler. The BLISS–16 compiler is a **cross-compiler**, that is, it executes on a VAX, a DECsystem–10, or a DECSYSTEM–20 but compiles code for its target system, the PDP–11. The BLISS–32 and BLISS–36 compilers are **native**: they execute on their own target system. Each BLISS compiler is described in a separate user manual that is dedicated to its dialect.

BLISS–16, BLISS–32, and BLISS–36 are dialects of a single language. Each dialect consists of a body of identical language features called Common BLISS (which forms the bulk of each dialect), plus a number of features either unique to one dialect or shared by only two of the three. Common BLISS constitutes the transportable language base. The dialect-specific features reflect architectural characteristics of one target system that are not found in each of the others,

for instance byte-addressing capability, found in the 16- and 32-bit target systems but not in the 36-bit systems. While it is possible to implement most programs in Common BLISS only, without reference to system-specific functions or characteristics, it is not always desirable to do so. This point is discussed further in *Section 1.5, "Program Transportability"*.

1.2. Language Objectives and Characteristics

This section discusses BLISS design objectives, and provides an overview of the language.

1.2.1. Design Objectives

Because of the system-software orientation of BLISS, a number of its primary objectives differ from those of application-oriented languages such as COBOL, FORTRAN, and PL/I. Foremost among those objectives are the following:

- Highly optimizable object code

- Simple and consistent facilities for operating on addresses
- Control constructs which encourage well-structured source code, in the interests of program reliability, clarity, and maintainability
- Facilities for defining both the representation of a user-designed data structure and the manner of accessing the data in that structure
- Optional access to specific features of the target-system hardware or operating system
- Facilities for defining, at an appropriately high level, the linkage conventions used in calling routines or procedures

Because the language supports three different computer systems, an additional objective is program transportability across the target systems. BLISS, therefore, includes many features specifically designed to facilitate transportable programming. These features are discussed in *Section 1.5, "Program Transportability"*.

1.2.2. Language Overview

BLISS has many of the features of other modern high-level languages. It has block structure, an automatic stack, and mechanisms for defining and calling recursive routines. It uses algebraic notation for calculations and has operations for arithmetic, shifting, comparison, and logic. It provides a variety of predefined data structures and permits the programmer to define additional data structures. It has facilities for testing and iteration that support clear and reliable programming. These same facilities also allow the compiler to perform extensive flow optimizations.

On the other hand, BLISS omits certain features of other high-level languages. It does not have built-in facilities for input/output, because a system-software project usually develops its own input/output or builds on basic monitor I/O or screen management services. It avoids certain kinds of automation of the programming process that introduce inefficiency for the sake of convenience. It is machine dependent to the extent that it permits access to machine-specific features, because system software often requires this.

BLISS has characteristics that are unusual among high-level languages. A name representing a data segment (that is, a storage location) is uniformly interpreted as the address of that segment rather than the value of the segment, and the language includes an explicit fetch operator that denotes "contents of".

Also, BLISS is an "expression language" rather than a "statement language". This means that every construct of the language that is not a declaration is an expression. Expressions produce a value as well as possibly causing an action such as modification of storage, transfer of control, or execution of a program loop. For example, the counterpart of an assignment "statement" in BLISS is, strictly speaking, an expression that itself has a value. The value of an expression can be either used or discarded in BLISS. When the value of an expression is discarded, the expression is said to be used in a "statement-like" way, that is, used solely for the action or side effect that it produces. See *Section 1.4.5, "Expressions"* for further discussion.

Finally, BLISS includes a macro facility that provides a level of capability usually found only in macro-assemblers.

1.3. Program Development

The typical development of a BLISS program, from inception to successful execution, is outlined below to introduce certain concepts and terms used later in this manual:

1. **Design –** To provide a logical structure for the program, you organize it into a set of **routines** and associated data structures. In general, each routine corresponds to a clearly identified, relatively independent function or subfunction of the program. One of the routines is the main routine. Later, when the program is executed, this routine is called by the operating system. The main routine controls the overall flow of the program, calling other routines, which may in turn call yet other routines, and so on, until every routine has done its assigned job.

2. **Programming –** Once the routines and data structures have been designed, they are programmed in the BLISS language. The routines are grouped into **modules** for the purposes of compilation. The routines grouped into a given module might, for example, consist of those programmed by one member of a project team. They might also reflect a logical grouping that aids overall system understanding and facilitates structured testing. Each module is a text file that is called a BLISS source file.

3. **Compilation –** Once the modules have been programmed, each module is compiled. Each module can be compiled individually; this is one practical advantage of dividing a large program into several modules. The result of each compilation is an object file. An object file is a sequence of encoded machine instructions and linker directives that is equivalent to the corresponding source module.

4. **Linking –** When all the modules of a program have been compiled, they are linked. The linker effectively "binds together" the various object modules, supplies any routines requested from a common-routine library, and converts the compiler-encoded relative addresses to actual machine addresses. *Section 1.7.1, "System Components"* gives further details on the linker. The result of linking is a single file that contains the executable program image.

5. **Execution –** The program image is executed. The first executions are normally done with the assistance of a debugger. As bugs are found, the development process cycles back to compilation, programming, or, most unfortunately, to design. Eventually, the program is ready for useful execution.

This manual provides the information necessary for the second step in the development process, programming. The BLISS user manuals (one for each dialect) provide complete information about the third step, compilation, plus guidelines for linking, executing, and debugging.

The user manuals also contain detailed information about certain dialect-specific features, such as machine-specific functions and module switches that describe the target-system environment, and about transportable programming.

1.4. The Main Features of BLISS

This section contains a brief description of BLISS. Those aspects of BLISS that are different from other high level programming languages are emphasized. The description is informal and omits many details; its purpose is to provide you with an intuitive understanding of BLISS that will be useful in further study of the language.

1.4.1. Data

All BLISS calculations are performed on values that correspond, in size, to the largest efficiently accessible unit of memory in each target system. This value, called a BLISS **fullword**, is 16 bits long for BLISS–16 (PDP–11 word), 32 bits long for BLISS–32 (longword), and 36 bits long for BLISS–36 (DECsystem–10/20 word). A fullword can be viewed as a sequence of single-bit logical values (true or false), as a sequence of ASCII character codes, or as a unitary value. As a unitary value, it can be interpreted as a signed integer, an unsigned integer, or a memory address.

In many high-level languages, a specific interpretation or "type" is permanently associated with each program variable. For example, one variable might be declared as containing an address value while another contains an unsigned integer. In BLISS, however, an interpretation is not associated with a variable. Instead, the interpretation of the value is specified by the operator that is applied to it. For example, BLISS has three operators for equality: EQL, EQLU, and EQLA. These operators interpret their operands as signed integers, unsigned integers, and memory addresses, respectively.

To conserve storage, data is often stored in **fields**, which are units of data that are less than a fullword in length. One field of special importance in all three dialects is the **bit**, which can be used to store a single logical value. In both BLISS–16 and BLISS–32, the 8-bit **byte** can be efficiently accessed and manipulated, and used for instance to store an ASCII character. In BLISS–32, the 16-bit **word** (which is the fullword of BLISS–16) can also be manipulated efficiently by the target hardware. No matter what field size is involved, however, a field value is always extended to a fullword value whenever it is fetched from memory.

1.4.2. Memory Addressing

Although calculations are always performed on fullwords, memory is addressed in fullword units only in the case of BLISS–36, where the target system's addressable unit is the full machine word. In both BLISS–16 and BLISS–32, the basic addressable unit is the byte. That is to say, if a memory address is incremented by 1 in either of these dialects, the location pointed to by the resulting address value is the next byte, not the next fullword.

Therefore, in order to precisely describe the interpretation of an address expression such as X+8 in a dialect-specific fashion, several different formulations would be required for the same expression. For example, assuming a fullword-reference context, the interpretation of the expression X+8 for BLISS–16 or BLISS–32 would be "Locate the fullword of memory that begins eight bytes after the byte whose address is X"; whereas the interpretation for BLISS–36 would be "Locate the fullword of memory that is eight fullwords after the fullword whose address is X".

In the interest of both generality and brevity, the nonspecific term "addressable unit" is used instead of "byte" or "fullword" in such descriptions, so that the two formulations given above reduce to the equivalent one: "Locate the fullword that begins eight addressable units after the unit whose address is X".

1.4.3. Fetching Values

In many programming languages, the interpretation of the name of a storage location depends on its context. For example, in FORTRAN, if the name appears as the left-hand side of an assignment, it represents the **address** of the storage location. If the name appears within an expression, it represents the **contents** of the storage location.

In BLISS, however, the interpretation of the name of a storage location does not depend on the context. Instead, the name always represents the *address* of the storage location. For example:

```
X+3
```

This expression is evaluated by adding 3 to the *address* that is associated with X. When the content of a storage location is needed, the fetch operator, a period (.), is used. For example:

```
.X+3
```

This expression is evaluated by adding 3 to the *contents* of storage-unit X. More exactly, the value of the expression is obtained as follows: Locate and fetch the fullword of memory that begins with the addressable unit whose address is X, and add 3 to the fetched value.

The fetch operator is an unusual feature of BLISS; it is not present in such languages as ALGOL, COBOL, FORTRAN, and PL/I. The omission of a fetch operator here and there is a frequent error among most beginning BLISS programmers. On the other hand, because BLISS always interprets a name as an address, it is easy to treat addresses as data, and address arithmetic can be performed in a simple and consistent way.

1.4.4. Assigning Values

A value is assigned to storage by the assignment operator, an equal sign (=). An example of an assignment is as follows:

```
X=2
```

This assignment means "Form a fullword value that represents 2, and then store that value in the fullword of memory whose address is X."

In BLISS, an assignment can be viewed as just another expression. Its first operand (left-hand side) provides a value that is interpreted as the address of a data segment. Its second operand (right-hand side) provides a value that is stored at the given address. The assignment expression itself has a value, namely the value of its second operand; more is said of this in *Section 1.4.5, "Expressions"*.

Often the left-hand side of an assignment is just a name. However, in BLISS there is no restriction on the expression that appears on the left-hand side of an assignment. Whatever that expression is, it is evaluated and the resulting value is interpreted as an address. For example:

```
X+6 = 2
```

This expression assigns 2 to the fullword of memory that begins six addressable units after the unit whose address is X. The example just presented is valid and illustrates an important feature of BLISS. However, such an assignment would not appear in a well-designed program, and especially not in a transportable one. Instead, an address computation, such as X+6 in the example, would be performed through a structure-reference (see *Chapter 11, "Data Structures"*).

1.4.5. Expressions

Many high-level programming languages classify each construct of the language either as a **statement**, which performs an action without producing a value, or as an **expression**, which calculates a value. For example, such languages classify the assignment construct as a statement, and do not permit its use in a context requiring a value.

In BLISS, any construct except a declaration can be used as an expression. For constructs that are statement-like, BLISS defines a value. For example, the value of an assignment is the value of the right-hand side of the assignment. The following expression contains an assignment:

```
2*(B = .C + 1)
```

When the expression is evaluated, it calculates 2*(.C+1). At the same time, without performing any additional calculation, it stores the value of .C+1 in location B.

The absence of statements from BLISS does not require a new approach to programming. Whenever a construct is used in a statement-like way, it is terminated by a semicolon and its value is discarded. The following expression is a **terminated expression**:

```
Q = 2*.R;
```

It assigns the value of 2*.R to Q and then, having no further use for the value, discards it. Such constructs as this, ending with a semicolon, play the role of statements in BLISS.

1.4.6. Blocks

A block is a syntactic feature of BLISS that is used to gather together a portion of a program and make it into a single unit (in fact, into a form of expression). In its most familiar form, a block is the keyword BEGIN followed by a sequence of **declarations** followed by a sequence of *terminated expressions* followed by the keyword END. For example:

```
BEGIN
LOCAL TEMP;
TEMP = .X;
X = .Y;
Y = .TEMP;
END
```

This block contains one declaration and three terminated expressions. The declaration specifies that TEMP designates a storage location that will be used only during execution of the block. Each of the three terminated expressions is an assignment and, together, they exchange the contents of X and Y. The entire block is, itself, a primary expression.

Sometimes it is useful to provide a value for a block. In that case, an expression without the terminal semicolon is placed at the end of the block. For example:

```
Z = BEGIN
    LOCAL TEMP;
    TEMP = .X;
    X = .Y;
    Y = .TEMP;
    .X EQL .Y
    END
```

This block exchanges the contents of X and Y just as the previous example of a block did. In addition, the contents of X and Y are compared and the value of the block is 1 or 0, depending on whether or not the values are equal. When execution of the block is complete, its value is assigned to Z.

In the first example, if the semicolon following the final expression (Y = .TEMP) were omitted, the block would have as its value the contents of location TEMP, according to the evaluation rule given for assignments in *Section 1.4.4, "Assigning Values"*. *Chapter 8, "Blocks and Declarations"* gives a full description of the semantics and use of the semicolon in the context of expressions and blocks.

A block that does not contain declarations is called a **compound expression**. An example that uses such a block is as follows:

```
IF .A NEQ 0
THEN
    BEGIN
    B = .P + .A;
    C = .Q + .A;
    END
```

In this example, the compound expression gathers two separate assignments into a single construct. Both assignments are performed if the contents of A is not 0 and both are skipped otherwise.

In BLISS, a parenthesis pair and a BEGIN-END pair can be used interchangeably. For example, the preceding example can be written equivalently as follows:

```
IF .A NEQ 0
THEN
     (
     B = .P + .A;
     C = .Q + .A;
     )
```

Or it can be written more compactly, as follows:

```
IF .A NEQ 0 THEN (B = .P + .A; C = .Q + .A;)
```

A block that uses a parenthesis pair and contains just one expression is a **parenthesized expression**; it is the ultimate specialization of a block. An example of the use of parenthesized expressions is as follows:

```
.(A + 1)*(B - 1)
```

Because the parentheses are present, the addition is performed before the fetch operation, and the multiplication is performed last of all. When the parentheses are removed, the expression appears as follows:

```
.A + 1*B - 1
```

This expression has a different meaning because the operators refer to different operands. According to the priority rules given in *Chapter 5, "Computational Expressions"*, the fetch operation is performed before the addition, and the multiplication is performed before the addition or subtraction. Thus, parenthesized expressions are used to override the priority rules.

1.4.7. Declarations

Every name in a BLISS program must be **declared**. The purpose of the declaration is to provide the BLISS compiler with information about the name. A simple example of a declaration is as follows:

```
OWN
     X;
```

This declaration says that X designates a storage location that is permanently allocated in the OWN program section before program execution begins. Note that, in the context of declarations, the semicolon is simply a mandatory terminator.

A more complicated example of a declaration is as follows:

```
OWN
     ALPHA: VECTOR[100] INITIAL(REP 100 OF (0));
```

This declaration not only specifies that ALPHA is an OWN name, but also gives two **attributes**, which begin with the keywords VECTOR and INITIAL. The VECTOR attribute describes the structure of the storage designated by ALPHA. The INITIAL attribute provides initial values for the storage.

The preceding examples are declarations of names of data addresses. An example of the declaration of a name of a routine address is as follows:

```
ROUTINE EXCHANGE(A,B): NOVALUE =
     BEGIN
     LOCAL
          TEMP;
     TEMP = ..A;
     .A = ..B;
```

```
    .B = .TEMP;
    END;
```

This routine exchanges the contents of the two locations that are given through the formal names A and B. The extra fetch operator used with these formal names reflects the fact that a formal name is the address of a storage location that contains a parameter; it is not the parameter itself.

The attribute NOVALUE indicates that this routine does not return a value, because the last expression within the routine body is a terminated expression. Therefore, a call on this routine must appear in a context that does not require a value. For example, the call could be used in a statement-like way. The semicolon following the keyword END is simply the required declaration terminator, and as such has nothing to do with whether or not the routine returns a value.

Some names do not represent addresses. For example:

```
MACRO
    Q = 0,3 %;
```

This declares the name of a macro, Q. During compilation, every occurrence of Q in the scope of this declaration is replaced by the text "0,3". Declarations are **scoped** by the block structure of a program. The same name can be used in different blocks for different purposes. Thus it is not necessary to use an awkward name because the appropriate name has been used in some other part of the same program.

1.4.8. Structures

The most commonly used forms of data structures are defined as part of BLISS. An example of such a structure follows:

```
OWN
    ALPHA: VECTOR[100] INITIAL(REP 100 OF (0));
```

In this declaration, VECTOR[100] is the **structure-attribute**. It specifies that ALPHA designates a data segment in storage that is not a single fullword, but rather is a sequence of 100 fullwords. The first of the fullwords is referenced by ALPHA[0], the second by ALPHA[1], and so on up to ALPHA[99]. An example of a reference to this vector follows:

```
ALPHA[.I-1] = 5
```

Suppose that, for a given execution of this assignment, the content of I is 8. Then the assignment is equivalent to the following:

```
ALPHA[7] = 5
```

Its effect is to set the eighth element of the vector to 5.

In addition to VECTOR, three other kinds of data structures (BITVECTOR, BLOCK, and BLOCKVECTOR) are defined as part of BLISS. Beyond that, however, is the capacity of BLISS to accept programmed definitions of data structures. This feature permits you to define data structures that are designed precisely for a given application. A part of the data-structure definition is the "algorithm" for accessing the structure. For example, a structure can be programmed to pack data in a way that saves storage or to include special checks for illegal accesses.

1.4.9. Flow of Control

Alternative actions to be taken by a program can be controlled by a **conditional-expression**. For example:

```
IF .X GTR 0
THEN
    Y = .X
ELSE
    Y = -.X;
```

This example sets Y to the absolute value of the contents of X. It ends with a semicolon, and is therefore a statement-like use of a conditional-expression. Another example follows:

```
Y = (IF .X GTR 0 THEN .X ELSE -.X);
```

This example also sets Y to the absolute value of the contents of X. However, in this example the value of the conditional-expression is used. Its value is .X or –.X, depending on whether or not the test is satisfied. Once the value of the conditional-expression is calculated, it is assigned to Y.

A more specialized construct for alternative flow of control is the **case-expression**. For example:

```
CASE .X FROM 1 TO 8 OF
    SET
    [1]:         REPORT1(.Z);
    [2]:         REPORT2(.Z);
    [4,7]:       Q = .Z+1;
    [INRANGE]:   ERROR1(.Z);
    [OUTRANGE]: ERROR2(.Z);
    TES;
```

The interpretation of this expression begins with the evaluation of .X; then, depending on the value of .X, one of five actions is taken. If the value is 1, the routine REPORT1 is called. If the value is 2, the routine REPORT2 is called. If the value is 4 or 7, the assignment Q = .Z+1 is performed. If the value is in the range from 1 to 8 but is none of the previous cases, then the routine ERROR1 is called. If the value is outside of the range 1 to 8, then the routine ERROR2 is called.

A third construct for alternative flow of control is the **select-expression**, which lies between the conditional-expression and the case-expression in its degree of specialization.

1.4.10. Loops

Iterative actions are controlled by **loop-expressions**. An example of the use of a loop-expression follows:

```
OWN
    SUM,
    LIST: VECTOR[21];
...
SUM = 0;
INCR I FROM 0 TO 20 DO
    SUM = .SUM + .LIST[.I];
```

The loop-expression in this example forms the sum of the 21 elements of the vector LIST. It does so by executing the assignment 21 times, once each for .I equal to 0, 1, 2, and so on through 20. In this example, the loop-expression is followed by a semicolon and is therefore used in a statement-like way. Note that the control parameters (0 and 20 in this case) can be any form of expression that has a value.

A second example of the use of a loop-expression follows:

```
OWN
    X,
```

```
    LIST: VECTOR[21];
...
X = (INCR I FROM 0 TO 20 DO
        IF .LIST[.I] EQL 0 THEN EXITLOOP .I);
```

The loop-expression in this example searches the vector LIST for an element that is 0. If a 0 is found, the value of the loop-expression is .I; that is, a value between 0 and 20 that shows where the 0 was found. If a 0 is not found, the loop runs to completion and the value of the loop-expression is (by definition) –1. In this example, the value of the loop-expression is used to provide, in a convenient way, for the case that there is no 0 in LIST.

1.4.11. Binding of Names

Most of the names in a BLISS program represent addresses – either data addresses or routine addresses. The operation of associating an address with a name is called **binding**. Once the name is bound, the use of the name becomes equivalent to the use of the address to which it is bound.

As an example of binding, consider the following use of the name BETA:

```
OWN
    BETA;
...
BETA = 4;
```

Suppose that BETA is bound to the address 1203. Then the assignment in the example is equivalent to the following:

```
1203 = 4;
```

In nearly all cases, you do not need to know the address to which a name is bound. Storage is allocated by the compiler, the linker, and the operating system, and you simply want references to storage to be consistent.

Occasionally, you may want to access a particular location. Suppose, for example, that a fullword used for communication with a certain input/output device is in location 80. Then that location can be set as follows:

```
BIND
    IOW = 80;
...
IOW = 0;
```

In this case, the assignment is entirely equivalent to the following:

```
80 = 0;
```

The use of the BIND declaration makes your intentions clear, not only to the reader but also to the compiler.

1.5. Program Transportability

Transportability of software is the use of the same source program in more than one system environment. The basis for transportable programming in BLISS is the extensive language base referred to as Common BLISS. In addition, BLISS provides many specific facilities that aid in achieving

transportability along with efficiency, either through parameterization of Common BLISS constructs, or conditional or compartmented use of dialect-specific code. The major facilities that support transportable programming are the following:

- Predefined data structures (for example, VECTOR, BITVECTOR, and BLOCK) that allow commonly used data structures to be allocated and accessed efficiently in each target environment.
- Predefined literals that reflect the parameters of the target architectures in terms of bits. These literals can be used, for example, to parameterize data declarations and storage references for greatest efficiency on each intended target system.

  A listing of the predefined literals and their values for each target system follows.

Name	Meaning	Value in:		
		BLISS–16	BLISS–32	BLISS–36
%BPVAL	Bits per BLISS value	16	32	36
%BPUNIT	Bits per addressable unit	8	8	36
%BPADDR	Bits per address value	16	32	18 or 30[1]
%UPVAL	Units per BLISS value	2	4	1

[1]Depending on the target-system CPU.

- User-definable data structures and named fields. The structure definition is a representation of the accessing algorithm, and it can make use of the predefined literals to provide field packing that is optimal for each target architecture.
- Character-string functions that permit efficient manipulation of string data regardless of the representation on the target architecture. Examples: CH$PTR creates a character-string pointer, CH $MOVE moves a character string, and CH$COMPARE compares the value of two strings. There are 25 such functions.
- Compile-time conditionals that allow compiled code to be explicitly different for different target architectures.
- A powerful macro facility that allows for different expansions for different target systems; for example, %BLISS32(BYTE) expands to its parameters (BYTE in this case) only if being compiled by the BLISS–32 compiler. Macros can also be used to segregate code sequences that differ for each architecture.
- REQUIRE and library files. Sets of common definitions can be kept in files that are selectively included in compilations through use of the REQUIRE or library declarations. This is a simple and efficient method of sharing common data structures and definitions between modules in a conditional fashion. It also permits compile-time conditionals and parameterized definitions to be maintained separately from the code in the modules.

1.6. Effects of Optimization

The semantic definitions of the BLISS language in this manual describe the useful, perceptible results of program execution as if those results were achieved without optimization of the object code. Wherever possible, then, the manual avoids discussion of how the results are actually obtained. The only exceptions are where a discussion of object code enables you to make a more efficient choice between several alternative constructs, for example, between two types of control expressions.

In particular, the optimization strategies employed by the compiler are not described. The optimizations reduce the cost of program execution by eliminating some of the actions defined by the language semantics, but they never affect the final results.

In some cases, however, the optimizations can be so extensive (global flow optimizations) that the object code generated does not show any obvious correlation to the corresponding sequence of source code. The degree of optimization performed by the compiler can be controlled by optimization switches, either in the module head (*Chapter 19, "Modules and Programs"*) or in the compiler command line. The BLISS user's guides describe the kinds of optimizations performed and the effect of the various optimization switches.

1.7. The BLISS Programming System

The **BLISS programming system** is the collection of software programs that supports the development of BLISS programs. Some of the components of the BLISS system are used only for BLISS programs; the compiler is an example. Other components are shared with other programming language systems; the linker is shared in this way.

Operating instructions for the compiler or the linker are not given here. Such instructions are essential (and are given in the appropriate BLISS user manual), but they never, or almost never, affect the results of program execution as described in this manual.

This section describes the components of the BLISS system and then discusses the evaluation of constant expressions by two components of the system, the compiler and the linker.

1.7.1. System Components

The BLISS system has five main components: the compiler, the linker, the operating system, the debugger, and a set of utilities.

1.7.1.1. Compiler

The **compiler** is specially written for the BLISS system (one for each dialect). It accepts a BLISS module as its input or source file. It produces an unlinked target-system program as its object file (although the compiler used for a given dialect may itself actually execute on another computer system, that is, it may be a cross-compiler). Because the compiler performs complicated and large-scale optimizations, the relationship between the source file and the object file is sometimes difficult to perceive; that is, it can be difficult to find the specific

instructions that implement a particular BLISS expression. Therefore, a plan for developing a BLISS program should involve as little reference to the object file as possible.

The compiler takes only one module at a time as its input. Therefore, the compiler cannot determine addresses that are used in the given module but declared in other modules; such addresses are external and must be left blank (unlinked in the object file). Furthermore, the compiler does not determine the absolute addresses of routines and data. Instead, the compiler expresses addresses as offsets relative to certain base addresses.

1.7.1.2. Linker

The **linker** is a target-system utility program that is shared by all of the programming languages for the target system. It accepts an unlinked object program, produced by the compiler, for each module of a program. It produces an executable program image as its output.

The linker finishes the job of preparing the program for execution. It has access to all modules of the program and can therefore fill in the external addresses. It can determine the required base addresses for routines and data and can therefore replace static offset addresses with absolute addresses.

1.7.1.3. Operating System

The **operating system** is a collection of target-system utility programs that are essential to any programming job. It includes a command that executes a program. This command loads the program image and starts execution. Thereafter, the operating system manages input/output, handles interrupts, and generally oversees program execution.

1.7.1.4. Debugger

The **debugger** is a program that assists you in finding errors in a program. The package includes features for dumping data in convenient representations and formats, for tracing data through the execution of the program, for establishing break points to halt program execution, and so on.

1.7.1.5. Utilities

The BLISS **utilities** are a collection of programs especially written to support the BLISS programming process. One such utility, for example, is the BLISS source-program formatter. The utilities are described in the BLISS user manual and in online documentation files available with each BLISS system.

1.7.2. Constant Expressions

When the value of an expression cannot change throughout program execution, it is a **constant expression**. Many important techniques for optimizing a program depend on the recognition and evaluation of constant expressions.

Some constant expressions can be evaluated as soon as they are written. For example, the value of the numeric-literal 52 is obviously fifty-two. Other constant expressions depend on addresses that are determined either by the compiler or by the linker. For example, the value of the expression X+6 depends on the address that is associated with X.

When the value of a constant expression is determined, the expression is bound. The process of associating values with constant expressions is a form of binding. These terms are most often applied to names; however, in BLISS a name is just a special case of an expression, and a bound name is just a special case of a bound expression. The main activity of the linker is to bind the names used in a program to appropriate addresses.

In certain contexts, BLISS requires a **compile-time constant expression**; that is, an expression that can be bound by the compiler. For example, when a VECTOR data segment is declared, its size must be given as a compile-time-constant-expression; this restriction permits the compiler to allocate storage for the data segment and thus avoid the expense of dynamic storage allocation.

Because the compiler does not determine absolute addresses, a compile-time constant expression usually cannot depend on a name that represents an address. The exception occurs in expressions such as X–Y or X EQLA Y; in these expressions, the offset addresses for X and Y (which are determined by the compiler) are sufficient to determine the values of the expressions.

In certain other contexts, BLISS requires a **link-time constant expression**; that is, an expression that can be bound by the linker. Since all addresses are determined by the linker, a link-time-constant-expression

can depend on a name that represents an address. Further details about both compile- and link-time constant expressions are given in *Chapter 7, "Constant Expressions"*.

Much of BLISS programming can be done without regard for the fact that a program goes through compilation and linking before it can be executed. The compile- and link-time constant expressions are important exceptions to this rule.

1.8. A Complete Program

An example of a complete program follows. The purpose of the example is to illustrate the overall structure of a BLISS program. The example is not a realistic program, although it is executable. A realistic program would require many pages for its listing as well as many pages of explanation. Instead, the example is a short program that reads a number from the terminal, adds 1 to it, and prints out the result.

The program is composed of two modules, TIO and E1. The first module, TIO, is assumed to be a general-purpose library module that performs input/output at the user's terminal. It includes an input routine, GETNUM, that reads a number that has been entered at the terminal, and an output routine, PUTNUM, that prints a given number at the terminal. The module TIO is not listed here. The second module, E1, is the specialized portion of the example program. It controls the entire process and performs the specified operation (the addition of 1) on the given data. This module is presented here.

```
MODULE E1 (MAIN = CTRL) =
BEGIN

FORWARD ROUTINE
    CTRL,
    STEP;

ROUTINE CTRL =

!+
!   This routine inputs a value, operates on it, and
!   then outputs the result.
!-

    BEGIN
    EXTERNAL ROUTINE
        GETNUM,     ! Input a number from terminal
        PUTNUM;     ! Output a number to terminal
    LOCAL
        X,          ! Storage for input value
        Y;          ! Storage for output value
    GETNUM(X);
    Y = STEP(.X);
    PUTNUM(.Y)
    END;

ROUTINE STEP(A) =

!+
!   This routine adds 1 to the given value.
!-
    (.A+1);

END
```

```
ELUDOM
```

An informal discussion of this module follows. Only the main features are mentioned, and some new terminology is introduced. The purpose is to give a general idea of how a module is constructed and how it works.

The module includes comments, each of which begins with an exclamation mark. Not included, however, is a long comment that normally appears at the beginning of a module and provides information about copyright, authorship, revisions, and so on.

The outer structure of the module is as follows:

```
MODULE E1 (MAIN = CTRL) =
BEGIN
...
END
ELUDOM
```

The first line gives the name of the module, E1. It also specifies that the main routine for the entire program is CTRL; therefore, when the program is executed, the operating system will call CTRL. The three dots represent the body of the module.

The body of the module begins with a forward-routine-declaration, which lists the names of the routines that are declared in the module. The remainder of the body is devoted to the declarations of the routines.

The first routine-declaration begins with the following line:

```
ROUTINE CTRL =
```

This line gives the name of the routine, CTRL. Because CTRL is not followed by a parenthesized list of names, the routine is not called with parameters. The purpose of the routine is to control program execution and to call other routines.

The body of the routine CTRL is given after the comment that describes the routine. It contains two declarations followed by three expressions. The declarations do not cause actions directly; instead, they describe the names that are used in the routine. The first declaration describes GETNUM and PUTNUM as names of routines that are declared in another module. The second declaration describes X and Y as the addresses of storage segments that are used during execution of this routine.

The three expressions are as follows:

```
GETNUM(X);
Y = STEP(.X);
PUTNUM(.Y)
```

The first two expressions are terminated (followed by a semicolon); the third is not. These expressions specify separate actions, and are executed (or more precisely, evaluated) one after another, in the order written. The first expression calls the routine GETNUM to read a number from the user's terminal and store it at address X. The second expression calls the routine STEP to add 1 to the contents of X and then assigns the result to Y. The values of the first two expressions are discarded; thus, these expressions are used in a statement-like way, solely for their side effects.

The third, non-terminated expression calls the routine PUTNUM to print the contents of location Y at the user's terminal, but also to provide a value for the routine as a whole. This is the value of the routine call, presumably a completion code returned by PUTNUM. One target operating system, VMS, requires such a value to be returned by the main routine. In the case of other target operating systems, the main-routine return value, if provided, is ignored.

The second routine-declaration begins with the following line:

```
ROUTINE STEP(A) =
```

This line gives the name of the routine STEP. It also gives a formal name, A, that represents the parameter of the routine. Because there is no NOVALUE attribute, this routine also returns a value. The body of the routine STEP is given after the comment that describes the routine. It is a single line, as follows:

```
(.A+1) ;
```

This line specifies that when this routine is called, the value it returns is calculated by adding 1 to the contents of formal location A, the value of the parameter. Observe that the semicolon here is the terminator of the routine declaration, and as such does not terminate the expression. It has no effect on whether or not the routine returns a value.

The expression that constitutes the routine body is enclosed in parentheses for added clarity; the effect would be exactly the same without the parentheses in this case. An equivalent way of expressing this routine declaration, which shows more clearly the role of the semicolon, is the following:

```
ROUTINE STEP(A) =

!+
!   This routine adds 1 to the given value.
!-

    BEGIN
    .A+1
    END;
```

Section 1.4.6, "Blocks" discusses the equivalence of the parenthesis pair and the BEGIN-END pair as used in these examples.

Chapter 2. Lexical Definitions and Syntax Notation

This chapter defines lexemes (the basic syntactic elements of BLISS) and the rules for the formation of valid BLISS source text. It also describes the syntax notation used in later chapters to define the larger constructs of the BLISS language.

The basic elements and rules of the language are as follows:

- *Characters and linemarks*. Characters are the indivisible units of program text. Linemarks serve to divide a character sequence into separate lines of source text. Together they constitute the lowest-level elements of syntactic structure.
- *Lexemes and spaces*. The lexemes of BLISS are analogous to the words and punctuation marks of ordinary English text. The spaces are used to separate lexemes where necessary and, optionally, to arrange the program text in a clear and attractive way. Together they constitute the next higher level of syntactic structure.

  Note that a *comment* in BLISS is simply a special form of a space from the lexical viewpoint.
- The *separation rules*. These rules govern the mandatory and optional use of spaces to separate lexemes.

Syntax notation, described in *Section 2.4, "The Syntax Notation"*, is used to formulate the syntactic rules that define the many constructs of the BLISS language. Each such construct consists of one or more lexemes. Thus these higher-level syntactic rules fundamentally depend on the separation rules for their formal interpretation, although the separations required and allowed by the syntactic rules are usually intuitively obvious without recourse to the separation rules.

2.1. Characters and Linemarks

At the lowest level of syntactic structure a BLISS module consists of a sequence of characters and linemarks. They are the smallest recognizable elements of the source text.

2.1.1. Characters

The characters that can appear in a module are listed and classified in the following table:

Category	**Characters**
Printing	
Letters	A B C . . . Z a b c . . . z
Digits	0 1 2 . . . 9
Delimiters	. ^ * / + – = , ; : ( ) [ ] < >
Special	$ _ % ! '
Free	" & ? @ \ ` { | } ~
Nonprinting	

	blank tab vertical-tab form-feed

All the characters in this table are members of the ASCII character set. However, the table does not include all the ASCII characters. Specifically, 30 of the 34 nonprinting ASCII characters do not appear in the table and must not be used in a BLISS module.

Note that this table shows which characters can be used in a BLISS program, but does not impose a restriction on data. BLISS data can use any ASCII characters. The characters that cannot be represented literally in the program text can be entered indirectly with numeric codes, through the %CHAR lexical-function described in *Chapter 15, "Lexical Functions"*.

2.1.2. Linemarks

The linemark is the separation between the end of one source line and the beginning of the next in a source file. On most terminals, you enter it into the file by pressing the RETURN, CARRIAGE RETURN, or NEWLINE key.

The linemark is represented in different ways in different target systems. On the PDP–11 and VAX systems, where a text file is a sequence of records, the linemark is represented by the division between two successive records. On the DECsystem–10/20, where a text file is a single character string, the linemark is represented by a line-feed, vertical-tab, or form-feed character; if any of these characters is immediately preceded by a carriage-return character, then that character is also part of the linemark.

2.2. Lexemes and Spaces

At the next higher level of syntactic structure a BLISS module consists of a sequence of lexemes and spaces. A lexeme is the smallest meaningful unit of the source text. Spaces are used to separate certain kinds of lexemes according to the separation rules, and are optionally used to separate other lexemes for greater readability and general formatting purposes. The division of a module into lexemes and spaces is especially important for the interpretation of macros, as described in *Chapter 16, "Macros"*.

2.2.1. Lexemes

The various types of lexemes that can appear in a module are listed and classified in the following table, with examples for each type except delimiters (single characters that are completely enumerated):

Category	**Examples**
Keywords	ROUTINE %ASCII AND
Names	
Predeclared	VECTOR MAX
Explicitly Declared	X BETA26 INITIAL_SIZE
Decimal Literals	0 23000
Quoted Strings	'ABC' 'He said, "Go!"' '77700'
Delimiters	
Operators	. ^ * / + – =
Punctuation Marks	, ; : ( ) [ ] < >

A delimiter serves either as an operator or as a punctuation mark. They are called delimiters because they separate neighboring lexemes without the use of intervening blanks. For example, the plus sign (+) delimiter can form the expression ALPHA+1 (consisting of three lexemes) without inserting blanks; however, using the keyword AND in place of the plus sign, without adjacent blanks, would form ALPHAAND1, which would be interpreted as a single lexeme.

2.2.2. Spaces and Comments

When two lexemes would otherwise run together to make a single lexeme, they must be separated by a **space**. A description of spaces is given in the following table:

Linemark	
Nonprinting Characters	blank tab vertical-tab form-feed
Comments	
Trailing Comment	! This is a program for entomologists.
Embedded Comment	%(Insert new routine here)%

The preceding table describes spaces informally, using two examples for the comments. A more precise definition is as follows:

- A **space** is a linemark, a nonprinting character (as listed in the table in *Section 2.1.1, "Characters"*) or a comment.
- A **comment** is a trailing comment or an embedded comment.
- A **trailing comment** is an exclamation character followed by the remainder of the line on which the comment begins.
- An **embedded comment** begins with the two characters "%(", followed by the text of the comment, followed by the two characters ")%". The text must not contain the sequence ")%", because that would prematurely end the comment; see guidelines below. An embedded comment can begin after any lexeme of a module and can extend to any later position in the module. However, an embedded comment must end in the same source file in which it began.

  Spaces are commonly used to arrange the module in a clear and attractive format and to insert comments on the workings of the program. However, when a module is translated by the compiler, the only role of spaces is to separate the lexemes of the module. For example, from the point of view of the compiler, a lengthy comment is equivalent to a single blank character.

2.2.2.1. Guidelines on the Use of Comments

Beginning with an exclamation point (!), a trailing comment anywhere in a source line is terminated by the next linemark (that is, by the end of the line in which it occurs). Thus, it is a generally safe and unambiguous form of comment and can be used, for example, to "comment out" a line of source text.

An embedded comment, beginning with the character sequence "%(", is terminated by the very next occurrence of the sequence ")%". This means that the embedded comment cannot be nested. Also, the sequence ")%" is a valid though ill-advised form of ending of a macro definition (see *Section 16.2, "Macro-Declarations"*). Thus an extensive embedded comment could be inadvertently terminated by the occurrence of ")%" in a macro declaration where the "%" character was intended to terminate a macro definition. For these reasons the embedded comment should be used with care. Also, avoid using it to comment out a body of code.

2.3. The Separation Rules

The use of spaces between the lexemes of a module is governed by the **separation rules**. The rules are as follows:

1. One or more spaces must appear between two lexemes if each lexeme is any one of the following:

   - A name
   - A keyword
   - A decimal-literal

   This rule requires the use of spaces wherever two lexemes would otherwise merge to form a single, longer lexeme.

2. One or more spaces can appear between any two lexemes. This rule permits the use of spaces to control format and provide comments.

3. A space must not be inserted into a lexeme. This rule prevents a lexeme from being broken into two lexemes. Some apparent exceptions arise in the case of a quoted-string lexeme, as described in *Section 4.3.2, "Restrictions"*.

2.4. The Syntax Notation

The **syntax** of BLISS is a collection of **syntactic rules** that describe the construction of a module (the unit of compilation). The special notation used for the syntactic rules is defined in this section.

Each syntactic rule defines a **syntactic name**. The syntactic rules are **interdependent**; that is, many of the rules define a syntactic name in terms of other syntactic names. However, the rules do not form a vicious circle of definitions because some of the rules define syntactic names directly in terms of **syntactic literals** (without reference to other syntactic names).

The ultimate syntactic name is **module**, which is defined in the syntactic rules given in *Chapter 19, "Modules and Programs"*. The description of the language begins with the definition of the syntactic name *expression*, in *Chapter 4, "Primary Expressions"*.

2.4.1. Syntactic Rules

A *syntactic rule* is divided into two parts by a vertical line. To the left of the line is the syntactic name that is defined by the rule; to the right, a string definition. In the simplest rules, the string definition is a single character or a single syntactic name.

In more complicated rules, string definitions are combined to make larger string definitions as follows: by concatenation (the joining of strings), by disjunction (the choice between two strings), or by iteration (the joining of several copies of a string).

An example of the simplest possible kind of rule is as follows:

```
dollar                  $
```

In English, this rule reads: "The syntactic name *dollar* designates the dollar sign ($) character." Note that the character is a syntactic literal, as defined in the following section; thus this rule completely defines

the syntactic name *dollar*, without reference to any other rules. Sometimes it is useful to give the same definition for several syntactic names. In such a case, the several names are written one above another and are joined by a brace.

```
{ position }
{ size     }        expression
```

In English, this rule reads: "The syntactic names *position* and *size* each designate an *expression*."

2.4.2. Syntactic Names and Syntactic Literals

A *syntactic name* is one or more English words composed of lowercase letters and connected by hyphens. Four examples of syntactic names are given in the two syntactic rules above, namely: dollar, position, size, and expression.

Further examples of syntactic names are as follows:

module
own-item
forward-routine-declaration
compile-time-constant-expression

Every syntactic name has at least two characters.

A *syntactic literal* is a printing character that is interpreted as itself when it occurs in a string definition. All printing characters are syntactic literals except the following:

- A character that is part of a syntactic name
- A brace character, { or }, or a vertical bar, |
- A period or comma that is part of the sequence " . . . " or the sequence ", . . . "

In this manual, syntactic names are always in lowercase and BLISS keywords are in uppercase.

2.4.3. Concatenations

A **concatenation** is a string definition composed of a sequence of two or more string definitions. If the definitions are adjacent (without intervening spaces), then the strings they define must also be adjacent. If the definitions are separated (by spaces), then the strings they define may or may not require separation, depending on the *separation rules* given in *Section 2.3, "The Separation Rules"*.

An example of a syntactic rule that uses adjacent concatenations is as follows:

```
volatile-attribute        VOLATILE
```

In English, this rule reads: "The syntactic name *volatile-attribute* designates the following string: the keyword VOLATILE." Because the eight letters VOLATILE (each one a syntactic literal) are adjacent in the rule, they must also be adjacent in the program.

An example of a rule that uses both adjacent and separated concatenations is as follows:

```
exitloop-expression        EXITLOOP exit-value
```

In English, this rule reads: "The syntactic name *exitloop-expression* designates the following string: the keyword EXITLOOP, followed by an *exit-value*."

In the English reading of any syntactic rule, the phrase "followed by" is an abbreviation for "followed by the spaces (if any) that are required by the separation rules, followed by."

2.4.4. Disjunctions

A **disjunction** is a string definition that permits a choice of one string definition from a set of several string definitions. The set of definitions is enclosed in braces. Each definition is separated from the preceding one by being on a new line or by a vertical-bar character.

The following is an example of a disjunction in which each choice is written on a separate line:

```
case-label          { single-value
                    { low-value TO high-value
                    { INRANGE
                    { OUTRANGE
```

In English, this reads: "The syntactic name *case-label* designates one of the following strings: (1) a *single-value*, (2) a *low-value* followed by the keyword TO followed by a *high-value*, (3) the keyword INRANGE, (4) the keyword OUTRANGE."

An example of a disjunction in which the choices are separated by vertical-bar characters is as follows:

```
octal-digit         { 0 | 1 | 2 | . . . | 7 }
```

In English, this reads: "The syntactic name *octal-digit* designates one of the following characters: 0, 1, 2, and so on to 7". Observe that once the set of choices is clearly implied, the ellipsis symbol (. . .) is used to indicate other choices. In some disjunctions, one of the choices may be the omission of a construct; in such a case, the word "nothing" is included in the braces. An example of a disjunction that uses the word "nothing" as one of the choices is as follows:

```
leave-expression    LEAVE label { WITH exit-value }
                                { nothing       }
```

2.4.5. Replications

A **replication** is a string definition that represents a sequence of one or more copies of a given string definition. The replication is indicated by writing an ellipsis symbol (. . .) after the given definition. The separation between the defined strings is determined by the separation rules, just as for concatenation.

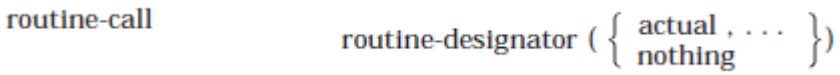
An example of a replication is as follows:

```
own-item            own-name { : own-attribute . . . }
                             { nothing               }
```

In English, this rule reads: "The syntactic name *own-item* designates the following string: an *own-name* followed by an optional *own-attribute-list*. An *own-attribute-list* is a colon followed by a sequence of one or more *own-attributes*". The extra syntactic name, *own-attribute-list*, is introduced only for the sake of the English reading.

A special kind of replication is indicated by writing a comma followed by an ellipsis symbol (, . . .) after the definition. The symbol means that each copy of the given definition is separated from the preceding one by a comma.

An example of a replication that uses the symbol follows:



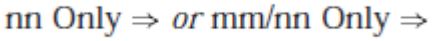
In English, the rule reads: "The syntactic name *routine-call* designates the following string: a *routine-designator*, followed by an open parenthesis followed by an optional *actual-list*, followed by a close parenthesis. An *actual-list* is a sequence of *actuals* that are separated from one another by commas". The extra syntactic name, *actual-list*, is introduced only for the sake of the English reading.

Note that whether or not the comma is included with the ellipsis symbol, the optional replication applies only to the string definition that immediately precedes the replication symbol.

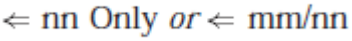
2.4.6. Dialectal Differences

Some of the syntactic rules given in this manual apply to only one or two of the three BLISS dialects. That is, some of the rules are not part of Common BLISS. Further, certain of the string definitions given within some rules are dialect specific.

These dialect-specific features are indicated in the syntax diagrams by flags of the following form, which precede a rule (or a group of rules):



Or a flag of the following form that follows a string definition:



In each case, *mm* and *nn* identify the dialects to which the syntactic feature applies (that is, 16, 32, or 36).

An example of an entire syntactic rule that is dialect specific follows:

In English, the dialect flag means: "The following syntactic rule applies to the BLISS–16 and BLISS–32 dialects only".

An example of both a syntactic rule and a string definition within the rule that are dialect-specific follows:

In English, the left-pointing dialect flag "(32 Only" means: "The string definition LONG is valid only in BLISS–32 as an alternative within the rule for allocation-unit (which itself applies only to the BLISS–16 and BLISS–32 dialects)".

Chapter 3. BLISS Values and Data Representations

The range of data values permitted and the kinds of data representations available are important characteristics of a programming language. Because the BLISS language is a systems implementation language, its value and data representations are closely related to those directly provided or efficiently handled by the machine architecture of each target system.

This chapter describes the values and data representations provided by each BLISS dialect. Because the three BLISS target systems have substantially different architectures (word sizes, addressable units, character string representations, and so forth), portions of this chapter are, necessarily, system specific.

3.1. BLISS Values

BLISS provides a variety of written (source program) representations for values (binary, octal, hexadecimal, and so on). These are described in *Chapter 4, "Primary Expressions"*. The normal representation is decimal; that is, any number in a BLISS program and in this manual is interpreted as decimal notation unless otherwise indicated.

The values on which the object program operates, however, are represented as **bit strings**. The maximum-length bit string that is efficiently accessible by a given target system (that is, a "word" or "longword" depending on the system) is called a **fullword** in BLISS terminology. The length of a fullword, in bits, for each target system is indicated by the numeric portion of the name of the respective dialect: 16, 32, or 36.

A bit string that is shorter than a fullword is called a **field value**. Several field value sizes are of particular importance in BLISS, depending on the dialect in question:

- For all dialects – The *bit*, which is the smallest unit of storage.
- For BLISS–16 – The *byte* (8 bits), which is the basic addressable unit in PDP–11 and VAX systems.
- For BLISS–32 – The byte, as above, and the *word* (16 bits), which is the intermediate size addressable unit in VAX systems.

Fullword values and field values play contrasting roles in BLISS. Fullword values are used as the basis for all calculations. Fields are used to achieve compact storage for values that do not require the maximum-length bit string for their representation. The two kinds of values are discussed separately in the following sections.

3.1.1. Fullword Values

The fullword value is the fundamental data type of BLISS. Specifically, the result of evaluating any BLISS expression is a fullword value.

In some cases, a fullword value can be viewed as a bit string without a specific interpretation, as when a value is moved from one storage location to another without modification. In other contexts, the bits of a fullword value are given a specific interpretation. A fullword value can be interpreted as any of the following:

- A signed integer, represented in two's complement notation

- An unsigned integer
- A sequence of character positions, each of which contains a code for an ASCII character
- A sequence of logical values, each of which represents "true" or "false"
- A memory address

Other interpretations for a fullword value can be devised, but these are the interpretations that are built into the operations of BLISS.

The length of a fullword, in bits, is given in each BLISS dialect by the predeclared literal %BPVAL (bits per value), that is, 16, 32, or 36 for BLISS–16, BLISS–32, and BLISS–36, respectively. Using this literal, you can express the range of a fullword value for each of the interpretations listed above for all dialects, as follows:

- Signed integer, i:

  ```
  -(2**%BPVAL-1) i (2**%BPVAL-1)-1
  ```

  In BLISS–16, for instance:

  ```
  -(2**15) i (2**15)-1
  ```

- Unsigned integer, i:

  ```
  0 i (2**%BPVAL)-1
  ```

- ASCII character positions:

  2 in BLISS–16
  4 in BLISS–32
  5 in BLISS–36

- Sequence of logical (Boolean) values:

  ```
  %BPVAL
  ```

- Memory address:

  ```
  Full address space of each target system
  ```

A fundamental rule of BLISS is the following: The interpretation of a fullword value is supplied by the context in which the fullword value is used. A given fullword value can have one interpretation in one context and a different interpretation in another context.

In this respect, the BLISS language is similar to machine language and is different from most high-level languages. Both BLISS and the target-system hardware interpret a value according to the operation applied to it. In contrast, most high-level languages associate a specific interpretation (or "type") with each value, independent of its context.

The BLISS rule for interpreting fullword values allows you to work closely with the hardware and, accordingly, to write more efficient programs. At the same time, however, this rule permits programming errors to arise as a result of the misinterpretation of values.

As a basis for an example of the interpretation of a fullword value, consider the following assignment:

```
X= -1
```

This assignment sets the contents of X to the two's complement representation of –1; that is, a sequence of %BPVAL 1's. The two expressions that follow interpret the contents of X in different ways:

```
.X LSS 4

.X LSSU 4
```

Both of these expressions use a less-than operator to compare the contents of X to 4. They yield 1 or 0 depending on whether or not the contents of X is less than 4. However, according to the definitions given in *Chapter 5, "Computational Expressions"*, the operators interpret their operands in different ways, as follows:

- The LSS operator interprets its operands as signed integer values. It finds that the contents of X is –1 and is therefore less than 4. Accordingly, the value of the expression is 1.
- The LSSU operator interprets its operands as unsigned integer values. It finds that the contents of X is a large positive integer (namely, (2**%BPVAL)–1) and is therefore not less than 4. Accordingly, the value of the expression is 0.

Because the negative number was assigned to X, it might be assumed that the user of the LSSU operator is incorrect. In fact, however, both expressions are valid. The question of which is correct depends entirely on the intentions of the programmer.

3.1.2. Field Values

A field value is a bit string that is shorter than a fullword. Field values are used in the following ways:

- Some stored values are "packed" and occupy only part of a fullword.
- Some BLISS operators and literals have values that can be represented in less than %BPVAL bits.

Whenever a field value arises during program execution, it is extended to become a fullword and then the appropriate interpretation is applied. The rules for the extension of values follow.

3.1.3. Extending Values

A field value is extended to a fullword value by placing a sufficient number of bits at the left end of the given value to provide a total of %BPVAL bits.

The following discussion of value extension is largely oriented toward BLISS–16 and BLISS–32, because the target systems for these two dialects allow allocation of scalar data segments in smaller-than-fullword units. Hence, these dialects have an allocation-unit and an extension-attribute that can be used in data declarations. As will be seen in *Chapter 5, "Computational Expressions"* and *Chapter 11, "Data Structures"*, however, these syntactic features are closely related to field-selectors, which are common to all three dialects. To the extent, then, that field values can arise in BLISS–36 as well as in BLISS–16 and BLISS–32, the following discussion is equally applicable to all dialects.

A value can be extended in two ways, as follows:

- Unsigned extension uses a zero bit for each additional bit.
- Signed extension uses a copy of the sign bit (leftmost bit) of the given value for each additional bit.

The kind of extension is determined in either of two ways. First, in BLISS– 16/32, an extension-attribute (UNSIGNED or SIGNED) can be included in the declaration of a data segment name (see *Section 9.2,*

"The Extension-Attribute – BLISS–16/32 Only"). Second, a sign-extension-flag can be used in a field-selector (see *Section 11.2, "Field-References"*). When the kind of extension is not explicitly given by an extension-attribute or a sign-extension-flag, unsigned extension is assumed as the default.

BLISS–16/32 ONLY

As the basis for some examples of value extension, consider the following declaration, which is valid in BLISS–16 or BLISS–32:

```
OWN
    X: BYTE SIGNED,
    Y: BYTE;
```

Suppose the contents of both X and Y are as follows:

```
11111111 (binary)
```

The declaration of X as SIGNED implies that this value is –1, that is, the two's complement interpretation of the given bit string. On the other hand, the declaration of Y as UNSIGNED (by default, since no extension-attribute is given) implies that its contents is 255, that is, the unsigned interpretation of the given bit string.

These declarations are invalid for BLISS–36 because the target-system architecture does not permit storage allocation in units of less than %BPVAL bits, that is, less than a 36-bit machine word. Fetching and storing of field values can be performed, however, through the use of explicit field-selectors, as illustrated in a later example.

The sign interpretations come into play when the contents of X and Y are fetched. The evaluation of .X uses signed extension to produce the following bit string:

```
11111 . . . 1111111111 (binary)
```

This is the two's complement representation of –1 represented in 16 bits for BLISS–16 or 32 bits for BLISS–32. In contrast, the evaluation of .Y uses unsigned extension to produce the following bit string:

```
00000 . . . 0011111111 (binary)
```

This is the unsigned representation of 255. Therefore, the two results are different, and the following expression would be false (that is, the low bit would have the value 0):

```
.X EQL .Y
```

In BLISS–36 as well as BLISS–16 and BLISS–32, you would obtain identical results using the following analogous set of declarations and fetch operations:

```
OWN
    X,
    Y;
```

This declares X and Y as the names of fullword, scalar data segments. Assume that the low-order eight bits of both these fullwords are one-bits. Then the following fetch operation specifies a fetch of the low-order eight bits of location X *with signed extension*. Upon evaluation the expression produces the value –1, as in the example above, represented in %BPVAL bits:

```
.X<0,8,1>
```

In contrast, the following fetch operation specifies a fetch of the low-order eight bits of location Y *with unsigned extension*, which produces the value 255 in %BPVAL bits:

```
.Y<0,8,0>
```

3.2. Data Segments

During the execution of a BLISS program, values are stored in *data segments*. A data segment consists of one or more addressable units of memory. In its simplest form, a data segment contains a single value. In its more complicated forms, a data segment can contain many values of various lengths. The different kinds of data segments can be classified as follows:

```
Data Segments
      Scalars
      Structures
         Predeclared Structures
            VECTOR Structures
            BITVECTOR Structures
            BLOCK Structures
            BLOCKVECTOR Structures
         Programmed Structures
```

A scalar segment contains a single value, whereas a structure may contain any number of values. Each predeclared structure is a part of the definition of BLISS, and it is invoked by using one of the predeclared structure names (VECTOR, BITVECTOR, BLOCK, or BLOCKVECTOR) in the declaration of a data segment. A programmed structure is defined by the programmer and can be used to organize the contents of a data segment in any way.

3.2.1. Addressable Units and Units per BLISS Value

The three target-system families supported by BLISS differ in four respects having to do with their storage organization that affect the source-language syntax and semantics to some degree. These differences are as follows:

1. Maximum (or only) "word" size, already described as the BLISS fullword consisting of %BPVAL bits.

2. Smallest directly addressable unit of storage.

3. Number of addressable units per BLISS value (that is, per fullword).

4. Size of an address value.

The bit size of the smallest addressable unit is given by the predeclared literal %BPUNIT (bits per unit). Its value is 8 for both BLISS–16 and BLISS–32 byte-oriented target systems; and 36 for BLISS–36 word-oriented target systems.

The number of addressable units per BLISS value is the quotient of %BPVAL over %BPUNIT. This value is given by the predeclared literal %UPVAL (units per value). Its value is 2 for BLISS–16 (two bytes per PDP–11 word), 4 for BLISS–32 (four bytes per VAX longword), and 1 for BLISS–36.

The final difference is the number of bits required for a maximum address value, given by the predeclared literal %BPADDR. Its value is 16 for BLISS–16, 32 for BLISS–32, and 18 or 30 for BLISS–36, depending on the setting of the EXTEND module-switch. This value is usually less significant than the others, as its utility is limited to certain kinds of operations on addresses that are not commonly required.

The literals just described are used in the subsequent discussions of data-segment types.

3.2.2. Scalars

In BLISS–16 and BLISS–32, the storage occupied by a scalar segment depends on the **allocation-unit** that is associated with the segment. The allocation-unit is given in the declaration of the name of the segment and is one of the following keywords:

LONG (for 32 bits) – BLISS–32 Only
WORD (for 16 bits) – BLISS–16/32 Only
BYTE (for 8 bits) – BLISS–16/32 Only

When no allocation-unit is given, WORD is assumed in BLISS–16 and LONG is assumed in BLISS–32. In BLISS–36, only fullword scalar segments can be allocated.

The kind of extension used when the value of a data segment is fetched depends on the **extension-attribute** (BLISS–16/32 only) that is associated with the segment or the field-selector associated with the fetch operation. The extension-attribute is one of the following keywords:

UNSIGNED (for unsigned extension)
SIGNED (for signed extension)

When no extension-attribute or field-selector is given, unsigned extension is assumed.

The extension-attribute does not affect the amount of storage used for a data segment. Its only effect is on the way the value is extended to %BPVAL bits when it is fetched. It is valid to give an extension-attribute with a fullword data segment, but the attribute has no effect since the value is already %BPVAL bits long.

The following is an example of the declaration of a scalar segment:

```
OWN X;
```

This declaration describes a segment that is allocated permanently before execution begins (because it is OWN), that is named X, that is a scalar (because no structure-attribute is given), that occupies a fullword (because no allocation-unit is given), and that uses unsigned extension (because no extension-attribute is given).

The features of a data segment can be illustrated in a diagram. In the following, the declaration of X is given together with the diagram for the corresponding data segment:

Declaration	**Diagram**		
OWN X;	2360 X	15	(%BPVAL)

This diagram represents a data segment in a simple and abstract way; that is, it does not show the specific layout of the data in terms of the byte boundaries (where applicable), bit sequences, and addresses of storage. A more detailed notation is introduced in *Chapter 11, "Data Structures"*. The diagram represents the data segment as follows:

- The address of the data segment is given in two forms. The first form is an (arbitrarily chosen) integer, 2360, used by the hardware to locate the segment. The second form is the name, X, that is used by the program to designate the segment.
- The storage is represented by a box followed by a parenthesized expression. The expression shows how many bits of storage the box represents.

- The contents of the data segment is given as a literal, 15, written inside the box. It is this part of the diagram that changes as program execution proceeds.

In this example, the value of X is 2360 (the address of the data segment), whereas the value of X is 15 (the contents of the data segment).

BLISS–16/32 ONLY

The preceding example describes a scalar that occupies a fullword. Examples of scalars that, in BLISS–16 or BLISS–32, occupy a word and a byte are as follows:

Declaration	**Diagram**		
OWN Y: WORD;	1000 Y	28	(16)
OWN Z: BYTE;	2440 Z	18	(8)

In these examples, each data segment has the UNSIGNED extension-attribute by default. Thus the values fetched from Y are in the range from 0 to (2**16)–1, and the values fetched from Z are in the range from 0 to (2**8)–1.

An example of a scalar that has the SIGNED extension-attribute follows:

Declaration	**Diagram**		
OWN R: SIGNED BYTE;	3002 R	–5	(8)

The values fetched from R range from –(2**7) through (2**7)–1. Thus although R and Z (in the preceding paragraph) both occupy eight bits of storage, their values are interpreted differently when they are fetched.

For the purposes of the following discussions, in BLISS–36 scalar data-segment declarations can be thought of as having an implicit allocation-unit of %UPVAL value (that is, one addressable unit per segment), and an implicit UNSIGNED extension attribute.

3.2.3. VECTOR Structures

A vector structure is a sequence of scalar **elements**. The number of elements is the **extent** of the vector, and is given as part of the declaration of the segment name. The elements are numbered, with 0 for the first element, 1 for the second, and so on.

Each element of a vector has the same allocation-unit and extension-attribute. This information can be given as part of the declaration of the vector. If the allocation-unit is not given, the default is the same as for scalar segments (fullword allocation). If the extension-attribute is not given, unsigned extension is assumed (where applicable).

An example of a vector follows:

Declaration	**Diagram**		
OWN A: VECTOR[3];	5440 A[0]	28	(%BPVAL)
	A[1]	5	(%BPVAL)
	A[2]	133	(%BPVAL)

This declaration describes a segment that starts at address 5440 and is named A. The declaration gives the extent of the vector as 3 and so the vector has three elements. The declaration does not give an allocation-unit, so each element occupies a fullword.

A particular element is selected by a bracketed subscript expression. Suppose that the contents of a data segment named IND is 3, and consider the contrast between the following expressions:

Expression	Value
A[.IND-2]	5440+%UPVAL (the address of the second element)
.A[.IND-2]	5 (the contents of the second element)

BLISS–16/32 ONLY

An example of a declaration that gives both allocation-unit and extension-attribute follows:

Declaration	Diagram		
OWN B: VECTOR[3,WORD,SIGNED];	46046 B[0]	15	(16)
	B[1]	3	(16)
	B[2]	4	(16)

This declaration describes a segment that starts at address 46046 and is named B. It is similar to the segment named A, described in the preceding paragraph. However, the allocation-unit is given explicitly as WORD, and therefore each element of the vector occupies 16 bits. It follows that the vector occupies only six bytes of memory. Furthermore, the extension-attribute is given explicitly as SIGNED, and therefore, the fetched contents of an element of B is subject to signed extension.

An example of a vector of bytes follows:

Declaration	Diagram		
OWN C: VECTOR[4,BYTE];	221 C[0]	7	(8)
	C[1]	7	(8)
	C[2]	2	(8)
	C[3]	4	(8)

This data segment is a vector of four elements and occupies four bytes of memory. Since an extension-attribute is not given, UNSIGNED is assumed by default.

3.2.4. BITVECTOR Structures

A **bitvector structure** is similar to a vector structure. However, bitvector structures are designed especially to handle bit strings, and each element of a bitvector structure is a single bit.

An example of a bitvector structure follows:

Declaration	Diagram		
OWN STATUS: BITVECTOR[15];	1604 STATUS[0]	1	(1)
	STATUS[1]	1	(1)

	. . . (and so on, until)		
	STATUS[14]	0	(1)
	(not used)		(n)

This declaration describes a segment that has 15 elements and thus makes use of 15 bits of memory. The number of unused bits, n, in the data segment allocated for this structure would be one in BLISS–16 and BLISS–32 (byte allocation), and 21 in BLISS–36.

A bitvector starts at the low-order (rightmost) bit of its first addressable unit of storage. Thus in BLISS–16 or BLISS–32, STATUS[0] designates the low-order bit of the byte whose address is 1604, STATUS[7] designates the high-order bit of that byte, STATUS[8] designates the low-order bit of byte 1605, and so on.

In BLISS–36, where the structure is entirely contained in one word, the references STATUS[0] and STATUS[8] designate the low-order bit and the ninth bit "from the right," respectively, of word 1604. Note that bit-position numbering in BLISS is consistent across dialects: bit numbers increase from low order to high order, "right to left," regardless of the target-system hardware convention.

Neither an allocation-unit nor an extension-attribute can be used with BITVECTOR. The number of addressable units allocated is the smallest number of units that can accommodate the given number of bits. When the contents of an element of a bit vector is fetched, unsigned extension is always used.

3.2.5. BLOCK Structures

A **block structure** is a sequence of **components**. The block as a whole has a name, which is declared using the BLOCK structure-attribute. In addition, each component of a block has its own name. A block is declared with a *size* and, in BLISS–16 and BLISS–32, an **allocation-unit**. The size specifies the amount of storage required for the entire block. The allocation-unit determines the units in which the size is measured. The default allocation-unit is the same as for a scalar segment declaration (fullword allocation).

The individual components of a block can have different sizes. The way in which the size of each component is specified is given in *Chapter 11, "Data Structures"*. For purposes of the present discussion, it is sufficient to state that the size is determined when the program is written and cannot change during program execution.

Observe that a block differs from a vector in two ways. A block is less flexible than a vector because, in normal usage, the name of a block component is given explicitly when the program is written, whereas the subscript of a vector element can be calculated during program execution. On the other hand, a block is more flexible than a vector because the components of a block can have various sizes, whereas the elements of a vector must all have the same size.

An example of a BLOCK structure, using BLISS–32, follows:

Declaration	**Diagram**		
OWN ITEM: BLOCK[ITEMSIZE,BYTE];	33300 ITEM[FLG]	0	(2)
	ITEM[N1]	235	(14)
	ITEM[LOC]	17	(32)

This declaration describes a segment that starts at address 33300 and is named ITEM. The declaration gives the size of the block as ITEMSIZE. The diagram shows that the individual components are FLG

(two bits), N1 (fourteen bits), and LOC (32 bits). Because ITEMSIZE must be the total number of bytes used, the diagram implies that the value of ITEMSIZE should be 6.

The address of a component of the block is written exactly as it appears in the diagram. Consider the contrast between the following expressions:

Expression	**Value**
ITEM[LOC]	33302 (the address of the third component)
.ITEM[LOC]	17 (the contents of the third component)

3.2.6. BLOCKVECTOR Structures

A *blockvector structure* is a sequence of *elements* (as is a vector structure), but each element consists of a block. The number of elements is the *extent* of the blockvector, and is given as part of the declaration of the segment name. The elements are numbered, with 0 for the first element, 1 for the second, and so on.

Each element of a blockvector is a sequence of components (as is a block). Each component is a scalar and has its own name. Therefore, the combination of the blockvector name, the subscript of an element, and the name of a component is used to designate a single value.

In addition to the extent, an *element-size* and, if BLISS–16 or BLISS–32, an **allocation-unit** are given in the declaration of a blockvector. The element-size specifies the amount of storage for each element (that is, the block size), and the allocation-unit determines the units in which the element-size is measured.

The default allocation-unit is the same as for a scalar segment (fullword allocation). The storage required for a blockvector is the product of its extent and its element-size.

An example of a BLOCKVECTOR structure, using BLISS–36, follows:

Declaration	**Diagram**		
OWN Q: BLOCKVECTOR[2,QS];	6000 Q[0,FLAG]	5	(8)
	Q[0,VAL]	62	(28)
	Q[0,PTR]	0	(36)
	Q[1,FLAG]	25	(8)
	Q[1,VAL]	78	(28)
	Q[1,PTR]	23	(36)

The declaration of Q gives the extent as 2 and the element size as QS. According to the diagram, each element has three components, FLAG, VAL, and PTR. Since QS must be the total number of fullwords used by each element, the diagram implies that the value of QS should be 2.

Suppose that the contents of a data segment named I is 0, and consider the contrast between the following expressions:

Expressions	**Value**
Q[.I+1,FLAG]	6002 (address of component)
.Q[.I+1,FLAG]	25 (contents of component)

3.2.7. Programmed Structures

The predeclared structures discussed in the preceding sections provide the data structures usually required for system programming. To provide for other data structures, BLISS has a feature, the STRUCTURE declaration, that permits you to design and use your own data structures. This feature of BLISS is described in *Chapter 11, "Data Structures"* where, in addition, each predeclared structure is defined in terms of a STRUCTURE declaration.

3.3. Character Sequence Data

The representation of character data differs among the three BLISS dialects due to basic architectural differences. In spite of these differences, it is possible to think about character data in a single, uniform way that applies to all BLISS target systems and, more importantly, to write BLISS programs that behave the same way and give the same results on all BLISS systems, even though the results are achieved in significantly different ways at object level.

The BLISS features for handling character data in this common (that is, transportable) way involve some new terminology and a set of special character-handling functions; these features are described in detail in *Chapter 20, "Character-Handling Functions"*.

The representation of character data and, in particular, sequences of characters is described here in two ways. First, character sequences are described in a general way that includes only the aspects that are common to all BLISS target systems. Second, the representation of character sequences is described specifically for each BLISS target system.

3.3.1. General Character Representation

Loosely speaking, a character sequence is like a vector of character data elements. This analogy may be useful in understanding the following description of BLISS character sequences. Fuller detail is given in *Chapter 20, "Character-Handling Functions"*.

A **character code** is a sequence of bits that represents a character. Usually the ASCII encoding of characters is used in BLISS.

A **character position** is the storage for a single character code. For a given implementation of BLISS, the size of a character position is determined by two factors: the requirements of the character code and the organization of storage.

A **character position sequence** is a portion of storage that is used for one or more character positions. Such a sequence has a *first* and *last* position. For each position except the first, there is a *previous* position, and for each position except the last, there is a *next* position.

A **character data segment** is a character position sequence that is allocated as a single portion of storage. In the simpler applications of character handling, it is possible to treat each character data segment as a separate unit, containing a complete character position sequence and allocated in the same way as other data segments.

A **character pointer** is a value that designates a character position. Sometimes a character pointer is set to the first character position of a sequence and remains there, providing access to the entire sequence. In other cases, a character pointer is used to scan back and forth in a sequence, selecting one position after another. A character pointer can be correctly interpreted only by a character-handling function. It occupies a fullword.

The **length** of a character position sequence is the number of character positions in the sequence. The length of a sequence is *not* included as part of the sequence itself. To fully specify a character position sequence, both its length and a pointer to its first position must be given. Typically, the parameters of the character-handling functions occur in pairs, a length followed by a pointer.

3.3.2. Character Sequence Operations

The basic operations of character handling are the allocation of storage, creation of a pointer, moving of a pointer, fetching or storing of a character code, and comparison of character sequences. All these operations must be performed by means of the specific character-handling functions provided for this purpose. For example, the contents of a character position must always be *fetched* or *stored* by means of a character pointer that designates the character position. In contrast, a character pointer can be fetched or stored like any other fullword value (by means of the fetch-operator (.) or the assignment operator (=)).

Returning to the analogy with a vector of character data elements, the following correspondences can be established:

- A character code corresponds to the contents of an element of the vector.
- A character position corresponds to the storage for an element of the vector.
- A character position sequence corresponds to a contiguous sequence of elements of a vector (possibly but not necessarily the entire vector).
- A character data segment is the complete vector.
- A character pointer corresponds to the address of an element of the vector.

This analogy is inexact in the following ways:

- A character position need not correspond to an addressable unit of storage.
- A character pointer is not simply an address value.

As described below, these considerations apply specifically to BLISS–36.

3.3.3. BLISS–16 Character Representation

In BLISS–16 there are two character positions per fullword. Characters are allocated in storage with the leftmost character of the source string in the low-order (or rightmost) character position of the first or only fullword. Additional fullwords or bytes are allocated in ascending address order. For example, the source character string ABCDEFGH would be allocated as follows:

Diagram		
7000	/BA/	(16)
7002	/DC/	(16)
7004	/FE/	(16)
7006	/HG/	(16)

Note that the eight-character string ABCDEFGH can appear only in the context of a PLIT (a type of primary expression) because a string literal itself, as a primary expression, cannot exceed the capacity of

a fullword: two character positions in BLISS–16. See *Chapter 4, "Primary Expressions"*. The BLISS–16 representation is related to the general BLISS representation of character sequences as follows:

- A character code consists of eight bits.
- A character position is a byte of storage.
- A character position sequence is a contiguous sequence of bytes of storage with successive characters, considered from left to right, contained in successive bytes from lower to higher addresses.
- A character data segment is also a contiguous sequence of bytes of storage.
- A character pointer is the address of a byte.

3.3.4. BLISS–32 Character Representation

In BLISS–32 there are four character positions per fullword. Characters are allocated in storage with the leftmost character of the source string in the low-order (or rightmost) character position of the first or only fullword. Additional fullwords or bytes are allocated in ascending address order. For example, the source character string ABCDEFGH would be allocated as follows:

Diagram		
36014	/DCBA/	(32)
Diagram		
36018	/HGFE/	(32)

Note that the eight-character string ABCDEFGH can appear only in the context of a PLIT (a type of primary expression) because a string literal itself, as a primary expression, cannot exceed the capacity of a fullword: four character positions in BLISS–32. See *Chapter 4, "Primary Expressions"*.

The BLISS–32 representation is related to the general BLISS representation in the same way as in BLISS–16.

3.3.5. BLISS–36 Character Representation

In BLISS–36 there are five ASCII character positions per fullword or six SIXBIT character positions. Characters are allocated in storage with the leftmost character of the source string in the high-order (or leftmost) character position of the first or only fullword. Additional fullwords are allocated in ascending address order. For example, the ASCII string ABCDEFGH would be allocated as follows:

Diagram		
21005	/ABCDE/	(36)
21006	/FGH /	(36)

Note that the eight-character string ABCDEFGH can appear only in the context of a PLIT (a type of primary expression) because a string literal itself, as a primary expression, cannot exceed the capacity of a fullword: five character positions in BLISS–36. See *Chapter 4, "Primary Expressions"*. The BLISS–36 representation is related to the general BLISS representation of character sequences as follows:

- A character code consists of seven bits.

- A character position is a 7-bit field of a 36-bit word of memory.
- A character position sequence is a contiguous sequence of character positions with successive character codes, considered from left to right, contained in adjacent 7-bit fields beginning at any of the five character positions in a word and continuing toward positions in the lower order part of the word and then to the high order 7 bits of the next word, and so on.
- A character data segment is a contiguous sequence of 36-bit words.
- A character pointer is a special 36-bit value that consists of both address and position and size information describing the character position.

  In DECsystem–10 terminology, a character pointer is a byte pointer that, when used as the operand of an ILDB (increment and load byte) instruction, will fetch the character code value from the indicated character position.

3.4. Storage Organization

During the execution of a BLISS–compiled object program, storage consists of the following:

```
Storage
       Storage for the given program
           The stack
           The registers
           Storage for the first module
           Storage for the second module
          . . .
          Storage for the last module
       Other storage
```

The other storage includes the routines and data of the operating system, the run-time routines for BLISS, and the storage for programs other than the given program.

The stack, the registers, and the storage for each module are described in the following sections.

3.4.1. The Stack

The stack is used to store temporary data associated with the execution of the routines in a BLISS program. The stack is composed of *frames*. Upon entry to a routine, a frame is pushed on the stack for use in executing that routine.

Upon return from the routine, the frame is popped from the stack. A stack frame contains data segments of two kinds. Some of the data segments are declared as LOCAL or STACKLOCAL. Such segments are directly accessible from the program and are used for values that are needed only during the execution of the routine in which they are declared. The other data segments are allocated by the compiler and are not accessible from the program. These segments are used for such values as the return address of the routine or the intermediate results that are produced during the evaluation of an expression.

The declaration of LOCAL and STACKLOCAL names is described in *Chapter 10, "Data Declarations"*. The relation between a routine and the stack is further described in *Chapter 12, "Routines"*.

3.4.2. The Registers

The registers of BLISS correspond to the general registers of the target-system hardware. Each register contains one fullword value. Each of the registers is considered to be a single data segment.

The use of registers is normally determined by the compiler, not the program. Access to a register uses less time than access to ordinary storage; therefore, registers are often used to store the intermediate results and addressing indices of a calculation. Under special circumstances, registers can be accessed by the program.

The declaration of register names is described in *Section 10.7, "Register-Declarations"*.

3.4.3. Storage for a Program Module

A module uses four kinds of **program sections**. Each kind of program section has a special purpose, as follows:

- An **OWN program section** contains a data segment for each name that is declared OWN in the module. Such a data segment is permanently allocated. It can be accessed only from the module in which it is declared.
- A **GLOBAL program section** contains a data segment for each name that is declared GLOBAL in the module. Such a data segment is permanently

  allocated. It can be accessed from the module in which it is declared and in any module in which its name is declared EXTERNAL.
- A **PLIT program section** contains a data segment for each PLIT used in the module.
- A **CODE program section** contains a code segment for each routine that is declared in the module.

You can leave the management of program sections to the compiler; and in that case each module will have no more than one of each kind of program section. On the other hand, you can specify several program sections of the same kind for a module and can determine which data segments or routines are allocated in which program sections. The division of storage for a module into sections permits the operating system to manage storage effectively. For example, an OWN section need be present only when its associated module is being executed, whereas a GLOBAL section must be present more frequently. For another example, the PLIT and CODE sections are not modified during program execution and can therefore be regarded as read-only storage.

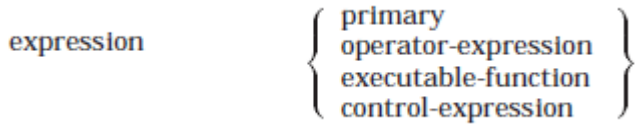
The declarations of OWN and GLOBAL segment names are described in *Section 10.1, "Own-Declarations"* and *Section 10.2, "Global-Declarations"*. The definition of PLITs is given in *Section 4.4, "PLITs"*. The declaration of routines is described in *Section 12.3, "Routine-Declarations"*.

Chapter 4. Primary Expressions

In most high-level languages, the term **expression** refers to the kinds of construct that perform calculation, such as the addition of two numbers or, perhaps, the concatenation of two strings. Such expressions have values; in fact, their sole purpose is to calculate values.

In BLISS, the term expression applies to all constructs of the language except declarations. For example, the construct that assigns a value to a data segment is an expression and has a value. As another example, the construct that controls an execution loop is also an expression and has a value. Thus it is possible, although unusual, to add the value of an assignment-expression to the value of a loop-expression.

There are four kinds of expression, as shown in the following syntax diagram:



This chapter describes **primary expressions**. It is the first of four chapters that describe the various kinds of expressions.

The first section of this chapter discusses primaries in a general way. Each of the remaining sections of this chapter describes one kind of primary in more detail.

4.1. Primaries

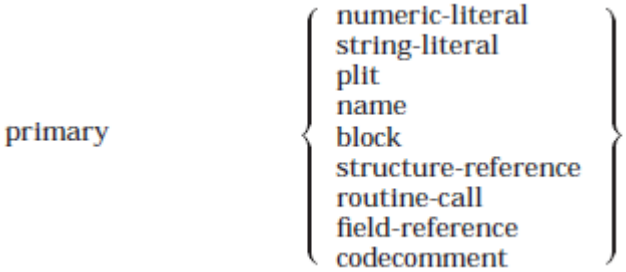
Every expression is built up from one or more primaries. The simplest form of expression is a single primary. More complicated expressions are constructed of primaries in combination with operators.

There is considerable variety among the primaries. A primary can be simply a numeric-literal, such as 4, or it can be a block of considerable length and complexity. A primary can specify a very elementary operation, such as the formation of a storage address, or it can call a long and complicated routine. The following are examples of primary expressions:

```
5                   !A numeric-literal whose value is 5

'Enter data:'       !A string-literal composed of 11 ASCII characters

PLIT (5,4)          !A pointer to a pair of literals

TOP_OF_LIST         !A name

F()                 !A call to routine F with no parameters

G(5, PLIT(5,4))     !A call to routine G with two parameters

X[ACCESS_LEVEL]     !A structure-reference to a field of a data
                    !structure named X

BETA<2,6>           !A field-reference to the six high-order bits of
                    !the byte at BETA
```

```
(.X + .Y)           !A simple kind of block, called a parenthesized
                    !expression

BEGIN               !A more complicated block, which contains a
LOCAL T;            !declaration and two expressions
T=0;
G(T,5);
END
```

4.1.1. Syntax



4.1.2. Semantics

The semantics of primaries is given in the following sections, where each kind of primary is considered individually.

4.2. Numeric-Literals

A numeric-literal is used to represent a specific number. An integer value can be written in any one of four radices: binary, octal, decimal, or hexadecimal. A special-purpose way of representing an integer is the character-code literal, which represents the ASCII code for a given character as a transportable, fullword value. A floating-point value can be written in single or double precision. Wherever the radix for a BLISS literal is not given, the radix is assumed to be decimal. This manual follows the same convention; that is, wherever a number appears in the text without an explicit radix, the number is assumed to be decimal.

The following examples show five different ways to write a numeric literal for the value 15.

```
15                  !Standard decimal-literal
%B'1111'            !Binary integer-literal
%O'17'              !Octal integer-literal
%DECIMAL'15         !Decimal integer-literal
%X'F'               !Hexadecimal integer-literal
```

The character-code-literal is used to express, in a transportable way, the numeric value of the ASCII code for a character. For example:

```
%C'A'
```

This has the decimal value 65, which is the ASCII code for character A.

Certain literal names are predeclared by the compilers and have specific numeric values. The values reflect various aspects of the target system architecture. For example, %BPADDR is predeclared with

a value that is the number of bits required for an address value, which varies for each target system. Therefore, the predeclared name %BPADDR has a different value for each BLISS compiler: 16 in BLISS–16, 32 in BLISS–32, and 18 or 30 (depending on the target-system environment) in BLISS–36. The predeclared literal names are described in *Section 14.1.5, "Predeclared Literals"*.

4.2.1. Syntax

numeric-literal { decimal-literal | integer-literal | character-code-literal | float-literal }

decimal-literal decimal-digit . . .

decimal-digit { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }

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

opt-sign { + | – | nothing }

integer-digit { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F }

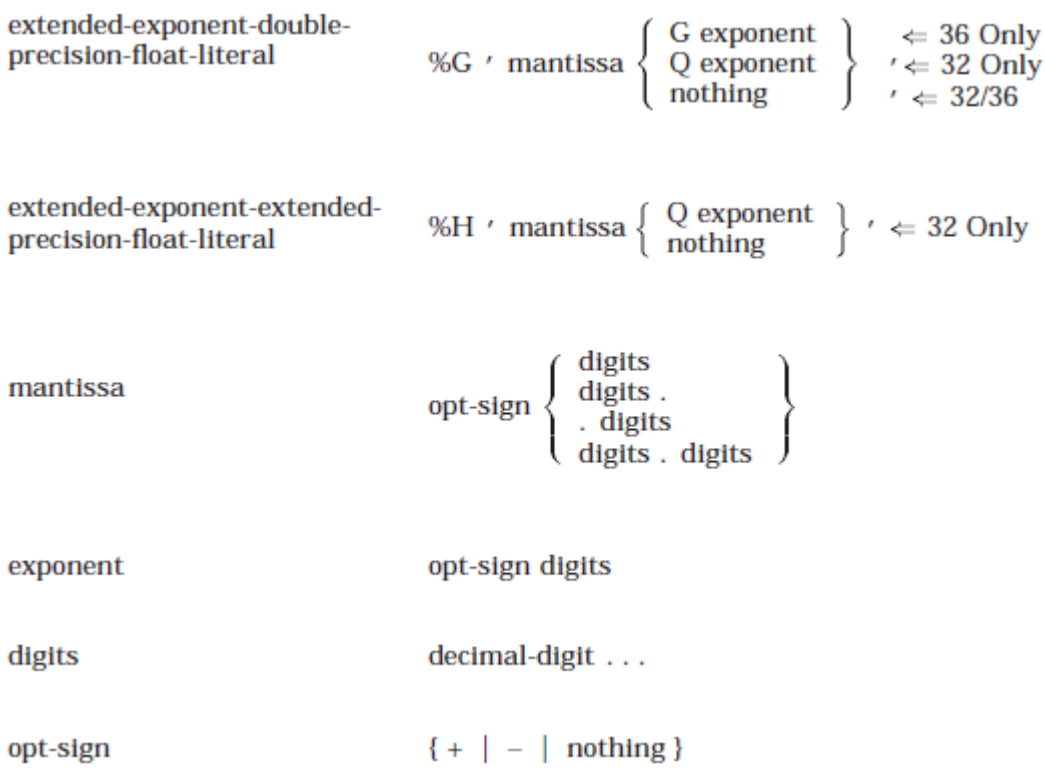
character-code-literal %C ' quoted-character '

quoted-character { printing-character-except-apostrophe | blank | tab | '' }

float-literal { single-precision-float-literal | double-precision-float-literal | extended-exponent-double-precision-float-literal | extended-exponent-extended-precision-float-literal }

single-precision-float-literal %E ' mantissa { E exponent | nothing } '

double-precision-float-literal %D ' mantissa { D exponent | nothing } '



Some of the numeric-literals are composed of two lexemes. Specifically, in an integer-literal, the radix indicator (%B, %O, %DECIMAL, or %X) is a lexeme and the remainder is another; and in a float-literal, the precision indicator (%E, %D, %G or %H) is a lexeme and the remainder is another. The quoted-string in a numeric-literal can be supplied by certain lexical-functions (see *Section 15.5, "Specific Lexical-Functions"*).

A *printing-character* is any ASCII character whose code, i, is in the range 33 i 126 (decimal). A *printing-character-except-apostrophe* is any printing character except an apostrophe. The *apostrophe* is the ASCII character with code 39 (decimal).

The *blank* is the ASCII character with code 32 (decimal). The *tab* is the ASCII character with code 9 (decimal).

4.2.2. Restrictions

The digits in an integer-literal must conform to the radix specified by the keyword at the beginning of the literal. Depending on whether the keyword is %B, %O, %DECIMAL, or %X, the digits must be binary, octal, decimal, or hexadecimal.

A space must not appear in a numeric-literal except between the lexemes of a two-lexeme numeric-literal (see *Section 4.2.1, "Syntax"*).

When a numeric-literal (other than a float-literal) is evaluated, its value, i, must fit in a fullword; that is, it must lie in the range

```
-(2**(%BPVAL-1)) i (2**(%BPVAL-1))-1
```

See *Section 3.1.1, "Fullword Values"* for the definition of %BPVAL for each target system.

When a float-literal is evaluated its value, x, must fit in the target system's machine representation of a floating-point value. The maximum approximate value range of x for each target-system family is as follows:

- For BLISS–16: 0.29*(10**–38) abs(x) 1.7*(10**38)
- For BLISS–32: 0.84*(10**–4932) abs(x) 0.59(10**4932)
- For BLISS–36: 0.56*(10**–308) abs(x) 0.9*(10**308)

The listed value ranges of x reflect %D for BLISS–16, %H for BLISS–32, and %G for BLISS–36.

Depending on the compiler used, float-literals can produce values that occupy up to four fullwords; therefore, float-literals producing values that occupy more than one fullword must appear in either a PLIT (see *Section 4.4, "PLITs"*) or an initial-attribute (see *Section 9.6, "The Initial-Attribute"*).

The relationship, by compiler, of float-literals to fullwords is as follows:

Float-Literal	**Size (Fullwords)**		
keyword	32	36	16
%E	1	1	2
%D	2	2	4
%G	2	2	–
%H	4	–	–

4.2.3. Defaults

The default for the sign of a numeric-literal is a plus sign (+). For example, the numeric-literal %O'777' is equivalent to %O'+777'.

The default radix is decimal; that is, when a sequence of digits appears without a radix keyword and without quotes, it is assumed to be a decimal-literal.

4.2.4. Semantics

A decimal-literal is interpreted as the decimal representation of an integer value.

An integer-literal begins with a keyword that determines its interpretation by giving the radix of the literal. Depending on whether the keyword is %B, %O, %DECIMAL, or %X, the sequence of digits within the quotes is interpreted as a binary, octal, decimal, or hexadecimal representation, respectively, of an integer value.

The value of a character-code-literal is the integer that is the ASCII character code for the quoted-character. When two apostrophes are used as the quoted-character, the value of the literal is the character code for a single apostrophe; that is, the character-code-literal %C"'" has the value 39 (decimal).

The evaluation of a numeric-literal produces an integer value. If the literal has a minus sign, then its value is represented as a negative number in two's complement form. The evaluation of a %E float-literal in 32 and 36 produces a dialect specific fullword value.

4.2.4.1. Limitations on Float-Literals

Note that values requiring more than %BPVAL bits for their representation cannot be stored in a fullword and cannot be directly operated upon by any of the BLISS operators or executable-functions.

Except for a few built-in machine-specific-functions, BLISS does not provide facilities for operating upon any float-literal as such. Float-literals are provided in BLISS in order to facilitate the development of special data segments and special routines for performing high-precision arithmetic.

4.3. String Literals

A string-literal contains a sequence of ASCII characters. The value of the string-literal is obtained by encoding the sequence of characters in one of several different ways, depending on the string-type of the literal (that is, %ASCII, %ASCIZ, %RAD50_11, %P).

A string-literal whose value occupies one fullword or less can be used as a primary, that is, can appear anywhere that a primary expression is allowed. The number of characters that can be encoded in a fullword varies with both the target system and the string-type (*Section 4.3.2, "Restrictions"*). Examples are:

```
%ASCII'AB'                  !in any dialect
%ASCII'ABCD'                !in BLISS--32 or BLISS--36
%RAD50_11'ABC'              !in BLISS--16 or BLISS--32
%RAD50_11'ABCDEF'           !in BLISS--32 only
%RAD50_10'ABCDEF'           !in BLISS--36 only
```

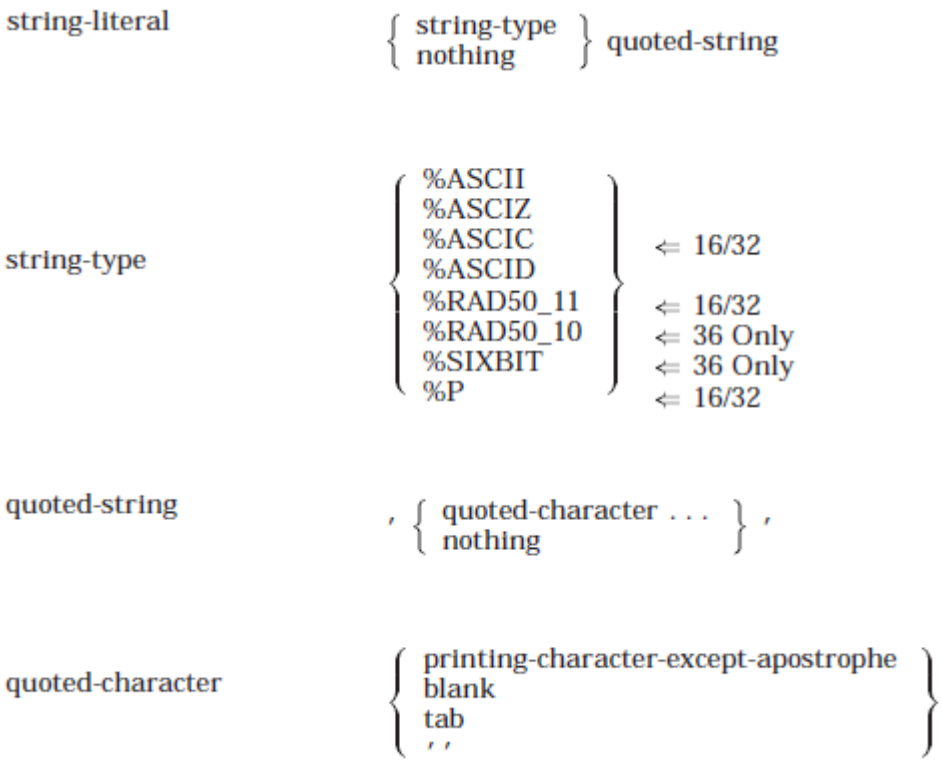
In each of these examples, the quoted string is encoded into one fullword or less in each of the dialects specified.

A string-literal whose value occupies more than a fullword is not a primary expression and can be used only within a PLIT expression (see *Section 4.4, "PLITs"*) or in an initial-attribute (see *Section 9.6, "The Initial-Attribute"*). An example follows:

```
'A complete list of errors follows:'
```

The encoded value of this string-literal, consisting of 34 character positions, occupies much more than a fullword on any target system.

4.3.1. Syntax



A *printing character* is any ASCII character whose code, i, is in the range 33 i 126 (decimal). A *printing-character-except-apostrophe* is any printing character except an apostrophe. The *apostrophe* is the ASCII character with code 39 (decimal).

The *blank* is the ASCII character with code 32 (decimal). The *tab* is the ASCII character with code 9 (decimal).

Some of the string-literals are composed of two lexemes, the string-type and a quoted-string. Spaces are permitted between the two lexemes.

The quoted-string in a string-literal can be constructed by certain lexical-functions, which are described in *Chapter 15, "Lexical Functions"*. A quoted-string constructed in that way can be composed of any sequence of ASCII characters and therefore is not restricted to printing characters, blanks, and tabs.

The quoted-string in a string-literal can also be supplied by another string-literal. This feature is mainly useful in the design of macros and is discussed in *Section 15.3.2.2, "Types of String-Literals"*.

4.3.2. Restrictions

A quoted-string is a single lexeme. As the syntax shows, the quoted-string can contain blanks and tabs. These characters are interpreted as characters in the string, not as characters that divide the quoted-string into several lexemes. Aside from blanks and tabs, no other *spaces* (as defined in *Section 2.2.2, "Spaces and Comments"*) can appear in the source text for a quoted-string.

A string-literal that is not a plit-string in a PLIT or initial-attribute must fit in one fullword. With %ASCID excepted, specific limitations on string length are given in the following table, by dialect and string-type:

Dialect	**Maximum Number of Characters in Fullword**					
	ASCII	**ASCIZ**	**ASCIC**	**RAD50_11**	**SIXBIT**	**RAD50_10 P**
BLISS–16	2	1	1	3	–	– 3[1]
BLISS–32	4	3	3	6	–	– 7[1]
BLISS–36	5	4	–	–	6	6 –

[1]Plus optional sign character.

BLISS–16/32 ONLY

A %ASCIC string-literal must contain no more than 255 quoted-characters.

A %RAD50_11 string-literal can contain only the characters A through Z, 0 through 9, blank, period (.), and dollar sign ($) in the quoted-string. Lowercase letters appearing in the quoted-string are encoded as the corresponding uppercase letters.

A %P string-literal must contain only the decimal digits (0 through 9) except for an optional initial sign (+ or –). There must not be more than 31 digits in the quoted-string.

BLISS–36 ONLY

A %RAD50_10 string-literal can contain only the characters A through Z, 0 through 9, blank, period (.), dollar ($), and percent (%) in the quoted-string. Lowercase letters appearing in the quoted-string are encoded as the corresponding uppercase letters.

A %SIXBIT string-literal may contain any quoted-characters except the following:

Character	Symbol	ASCII Code
Tab		9
Accent grave	`	96
Open brace	{	123
Vertical bar	|	124
Close brace	}	125
Tilde	~	126

The parenthesized ASCII codes are in decimal. Lowercase letters appearing in the quoted-string are encoded as the corresponding uppercase letters.

Other restrictions on the length of string-literals (if any) are given in the appropriate BLISS user manual.

4.3.3. Defaults

The default for the string-type is %ASCII. For example, the string-literal 'abc' is equivalent to %ASCII'abc'.

The default for the sign in a %P string-literal is "+". For example, the string-literal %P'2' is equivalent to %P'+2'.

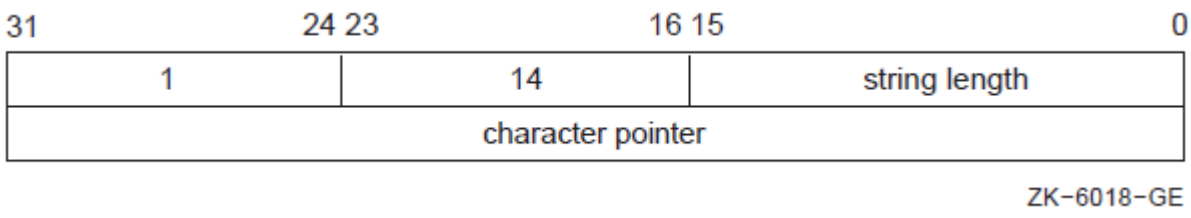
4.3.4. Semantics

Each quoted-character in a string-literal represents one character code in the value. A printing-character-except-apostrophe, a blank, or a tab represents itself. A sequence of two apostrophes represents a single apostrophe.

A %ASCID string-type is similar to a %ASCII type; however, %ASCID differs in that it creates a string descriptor for the quoted-string, and expands to the address of the data segment that contains the descriptor. The string and its descriptor are allocated in a PLIT program section (see *Chapter 18, "Special Features"*), and just as the value of a PLIT is the address of the plit-body, the value of %ASCID is the address of the descriptor.

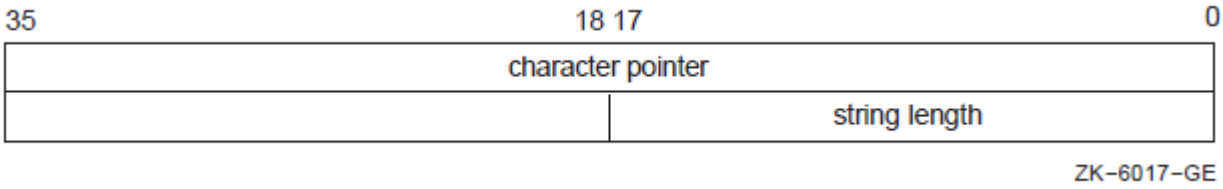
The %ASCID string creates the following descriptor formats:

For BLISS–32:

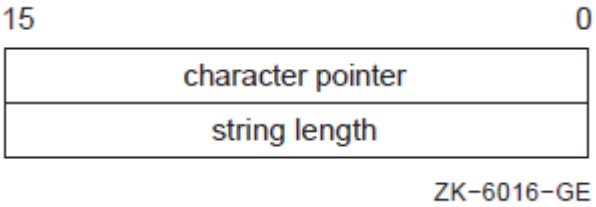


Note that *only* the BLISS–32 implementation of %ASCID is compatible with XPORT strings.

For BLISS–36:



For BLISS–16:



This format follows the PDP–11 Extended Instruction Set guidelines. Note that the string length must be an unsigned 16-bit quantity in the range 0 to 65535 decimal.

The remaining semantic description uses the generalized terms *character position* and *character position sequence*. The machine-specific equivalents of these terms are given in *Section 3.3, "Character Sequence Data"*. See also *Chapter 20, "Character-Handling Functions"*.

The value of a string-literal is determined in several steps, as follows:

1. For string-types %ASCIZ and %ASCIC, augment the string of quoted-characters as follows:

    a. If %ASCIZ, add a trailing null character (ASCII code 0) to the string.

    b. If %ASCIC (16/32 only), count the characters in the quoted-string and use this (8-bit integer) count as the initial character of the string, preceding the first quoted-character.

2. Encode the character string, augmented as required by step 1, according to the string-type and dialect, as follows:

    a. For string types %ASCII, %ASCID, and %ASCIZ, form a character position sequence that has one character position for each character in the string. For BLISS–16 and –32, use the 8-bit ASCII code of the ith character as the value of the ith character position. For BLISS–36, use the corresponding 7-bit ASCII code. For rules governing the filling of the last unit of storage see *Section 4.4.4, "Semantics"*.

    b. For string-type %ASCIC (16/32 only), form a character position sequence as in step 2.a, but use the initial count character value as is for the first character position.

    c. For string-type %RAD50_11 (16/32 only), extend the original quoted-string with enough trailing blank characters to make up a multiple of three characters, if necessary. Then use Radix–50 encoding to form a character position sequence that has two character positions for each group of three characters in the string. If necessary, extend the resulting character position sequence with enough trailing, zero-valued positions to fill the final (or only) fullword occupied by the sequence.

d. For string-type %RAD50_10 (36 only), use Radix–50 encoding to form a fullword for each group of six (or fewer) quoted-characters in the string. This encoding always produces one or more complete fullwords.

e. For string type %SIXBIT (36 only), form a character position sequence that has one (6-bit) character position for each character in the string. Use the SIXBIT code equivalent of the ASCII code of the ith character as the value of the ith character position. If necessary, extend the resulting character position sequence with enough trailing, zero-valued positions to fill the final (or only) fullword occupied by the sequence.

f. For string-type %P (16/32 only), use the PDP–11/VAX packed decimal string encoding to form a sequence that has one byte for each two digits of the quoted-string, and that provides a position for the sign in the last byte. Leading zero characters are not discarded in forming this sequence. The packed decimal encoding is described in the *VAX Architecture Handbook*.

Note

The ordering of character positions in storage is system dependent, and is described in *Chapter 3, "BLISS Values and Data Representations"*. The ASCII, Radix–50, and SIXBIT string encodings are described in *Appendix B, "String Encodings"*.

---

3. Use the character position sequence obtained in step 2 as follows:

   a. If the given literal appears in a plit or initial-attribute, use the sequence as the value of the literal.

   b. If the given literal does not appear in a plit or initial-attribute and the sequence is contained in a single fullword, the fullword is the required literal value.

   c. Otherwise, the sequence is invalid as a string-literal and the literal value is undefined.

The interpretation of a string-literal is performed entirely by the compiler. If the string-literal is a plit-string, then the compiler uses the value in forming a literal in PLIT storage, as described in *Section 4.4, "PLITs"*. If the string-literal is an initial-value, then the compiler uses the value to initialize the contents of a data segment, as described in *Section 9.6, "The Initial-Attribute"*. Otherwise, the compiler incorporates the value of the string-literal in the object code it is generating.

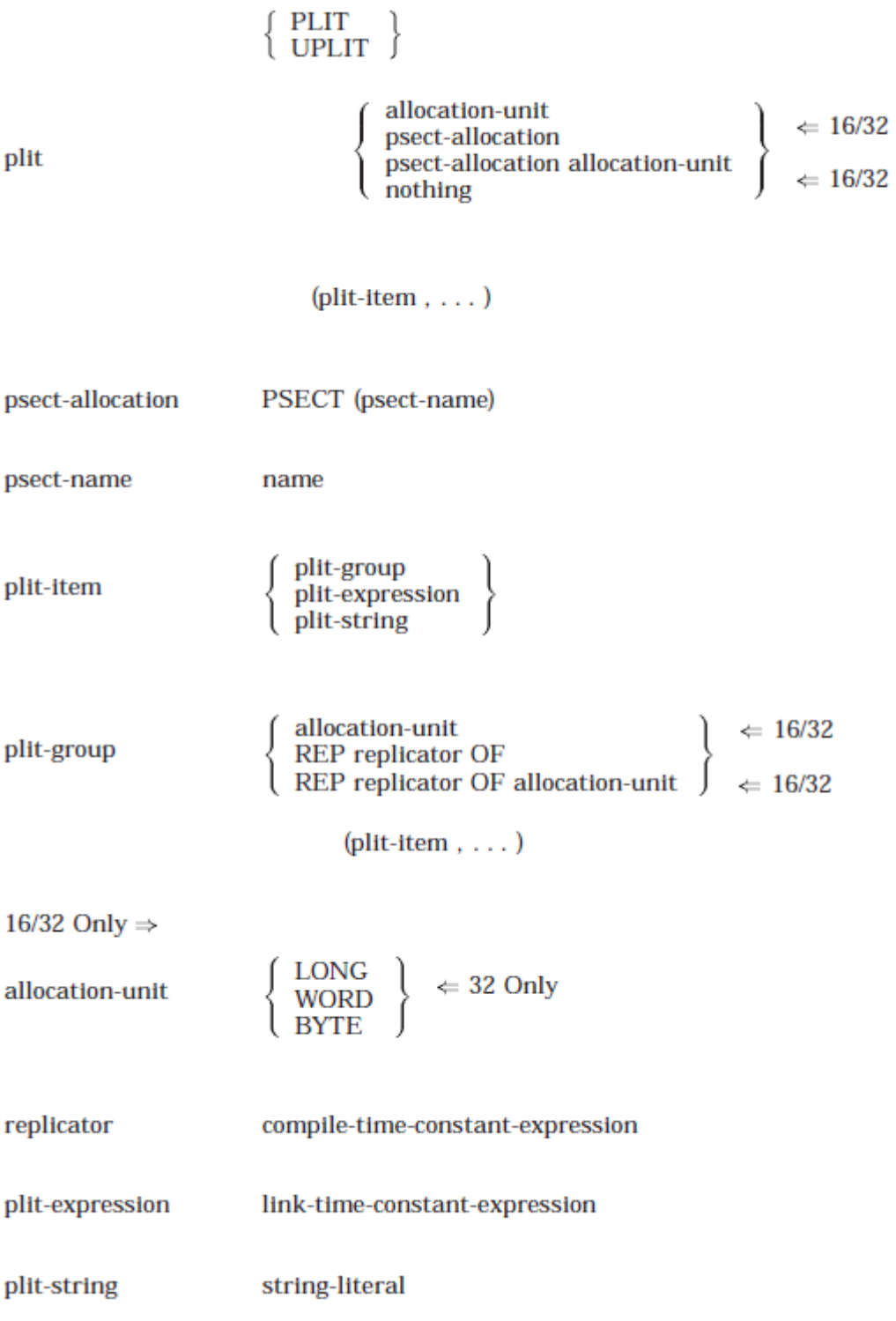
4.4. PLITs

A constant value that requires no more than a fullword of storage can be represented by a numeric-literal or string-literal that stands alone (that is, is not contained in a PLIT). A constant value that requires more storage must be represented by a PLIT.

The value of a PLIT is not the value of the given constant but rather the address of a data segment that contains the given constant. The data segment for a PLIT is allocated in a PLIT program section, and it is initialized to the given constant value before program execution begins.

There are two kinds of PLITS. The *counted* PLIT begins with the keyword PLIT, which stands for "Pointer to literal". The data segment for this kind of PLIT begins with an extra fullword that contains the *count* for the PLIT. The count is the number of fullwords in the PLIT excluding the fullword used for the count. The second kind of PLIT, the *uncounted* PLIT, begins with the keyword UPLIT, which stands for "uncounted pointer to literal." The data segment for this kind of PLIT does not include a fullword for the count.

4.4.1. Syntax



4.4.2. Restrictions

An appropriate psect-declaration (see *Section 18.1, "Psect-Declarations"*) must be made before a psect-allocation attribute (see *Section 9.8, "The Psect-Allocation Attribute"*) can be used in a PLIT.

The value of a replicator must not be less than zero.

BLISS–16/32 ONLY

The value of a plit-expression allocated as BYTE must lie in the range –(2**7) through (2**8)–1. The value of a plit-expression allocated as WORD must lie in the range –(2**15) through (2**16)–1.

4.4.3. Defaults

When no "REP replicator OF" construct is given, a replicator value of 1 is assumed.

4.4.4. Semantics

A PLIT causes constant data to be allocated. The value of the PLIT is the address of the first addressable unit of the data specified by the plit-items. The compiler determines an address offset for the PLIT, and the linker binds this offset to an absolute address.

If the PLIT has the keyword PLIT and therefore is a counted PLIT, then the count is located in the fullword preceding the data specified by the plit-items. The count indicates the number of fullwords occupied by the PLIT data.

In the simplest case, a PLIT is just the keyword PLIT or UPLIT followed by a parenthesized list of plit-expressions or plit-strings. In this case, values of the items are laid out in storage, starting at the PLIT address and continuing in the direction of increasing addresses. The value of each plit-expression occupies a fullword. The value of each string-literal occupies as many character positions as the string requires, with unused character positions added, if necessary, to fill out the final fullword.

BLISS–16/32 ONLY

When an allocation-unit is present, it specifies explicitly the unit of storage to be used. Depending on whether the allocation-unit is LONG, WORD, or BYTE, the value of each plit-expression occupies a longword, a word, or a byte, respectively. Similarly, the value of each string-literal occupies as many bytes as the string requires, with unused bytes added, if necessary, to fill out the last unit of storage. The allocation-unit LONG and the longword storage unit apply to BLISS–32 only.

When an allocation-unit is given, the item or items to which it applies are enclosed in parentheses. Several allocation-units can be used in a single PLIT; for any given item, the innermost allocation-unit is the one that applies.

When both a psect-allocation attribute and an allocation-unit of storage are used in a PLIT, they can appear in any order. For example:

```
PLIT PSECT( $OWN$ ) BYTE(7)
```

The psect-name (OWN in the example) specified in the attribute must be either predeclared, a default program-section name, or explicitly declared in a preceding psect-declaration. The psect-allocation attribute provides a more convenient way of making program-section assignments for a PLIT than is possible using the psect-declaration alone (see *Section 9.8, "The Psect-Allocation Attribute"*).

When a "replicator OF" construct is present, it specifies the repetition of the plit-group that follows it. The plit-group is evaluated before it is repeated. Thus, if the plit-group contains an embedded PLIT, the embedded PLIT is allocated once, and its address is used in each repetition of the plit-group.

The evaluation of PLITs is performed by the compiler, the linker, and the operating system before program execution. Thus during program execution, a PLIT represents the constant address of a sequence of constant values.

When the values specified by a PLIT do not completely fill the last fullword of the PLIT, the values of the unused character positions are undefined. A program that attempts to access the unused character positions is invalid.

PLITs are not necessarily allocated in the order in which they are written, and unused storage may be left between the storage for one PLIT and that for the next. Therefore, the relative positions of two PLITs are undefined. A program that depends on the relative positions of two PLITs is invalid.

4.4.5. Pragmatics

A plit-expression is not restricted to numeric-literals. It can be any link-time-constant-expression, and can therefore include address-valued names whose value is established at link time. Suppose the following declarations are given:

```
OWN
    A: VECTOR[10],
    B;
EXTERNAL
    X;
```

Then, within the scope of these declarations, the following PLIT can be used:

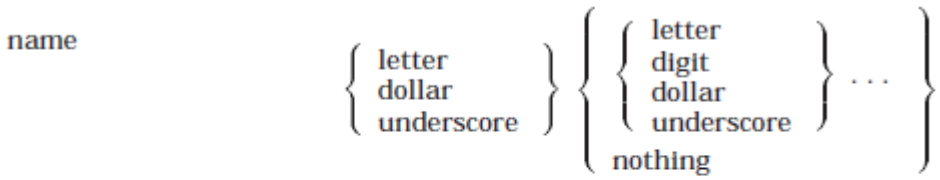
```
UPLIT(A[4], B+2, X)
```

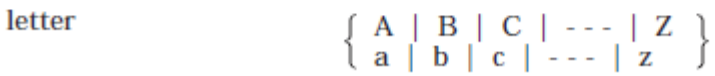
This PLIT occupies three fullwords. The first contains the address of the fifth element of A. The second contains the address B plus 2. The third fullword contains the address X.

4.5. Names

A name usually designates the address of a routine or a data segment. The value of such a name is determined by the compiler, linker, and operating system together. Within the scope of a given declaration of a name (as defined in *Section 8.2, "Declarations"*, the value of a name does not change during program execution.

4.5.1. Syntax





digit	{ 0 | 1 | 2 | - - - | 9 }
dollar	$
underscore	_

A name can be constructed by the %NAME lexical-function, described in *Section 15.5.4, "Name-Functions"*. A name constructed in this way can be composed of any sequence of ASCII characters and therefore need not satisfy the syntax given above.

4.5.2. Restrictions

A name must not be more than 31 characters long in any case.

The reserved keywords (listed in *Appendix A, "Predefined Identifiers"*) must not be used as names.

A name is a single lexeme and must not contain a space.

The dollar character is reserved for use in software supplied by VSI.

BLISS–16/36 ONLY

Names declared as global or external must be unique within their first six characters (throughout a program), to assure correct linking.

4.5.3. Semantics

When two names are compared, the distinction between uppercase and lowercase letters is ignored. Thus the following items are considered to be four instances of the same name:

```
BETA beta Beta bEta
```

This equivalence also applies to keywords. The only place where an uppercase letter is distinguished from a lowercase letter is in a quoted-string.

The interpretation of a name depends on its *declaration*. Declarations are described in *Chapter 8, "Blocks and Declarations"*.

4.6. Blocks

In its simplest form, a block is a means to gather together one or more expressions to form a single primary expression. In its more complicated forms, a block contains declarations and determines the scope of those declarations. It provides the fundamental large-scale unit of BLISS program structure.

For example:

```
5 * (.A + .B)
```

The block (.A + .B) serves to specify that the value of .A + .B is one of the operands of the multiplication operator.

The following block contains a declaration of local data segment T, which is used within the block as a temporary variable:

```
X = BEGIN
    LOCAL T;
    T = 2 + F();
    T = .T * G(.T);
    .T
    END
```

When the block is completed, the contents of T become the value of the block, and are assigned to X.

A complete description of blocks is given in *Chapter 8, "Blocks and Declarations"*.

4.7. Structure-References

When a data segment consists of a structure of several values, a structure-reference is used to fetch or store the individual values. A structure-reference can also be used to designate the address of a contained value.

Examples of expressions containing structure-references follow:

```
X=.A[.I]

TABLE[Q(.X+2)+3] = 5

F(ALPHA[FIELDNAME,.J-1])
```

The complete description of structure-references is given in *Chapter 11, "Data Structures"*.

4.8. Routine-Calls

A *routine-call* causes the execution of a routine. The called routine can be a part of the same module that calls it or it can be part of another module in the same program. The routine can be written in BLISS or in some other language that is supported by the target system.

The execution of a routine can have two kinds of effects. First, it can calculate a value that is returned as the value of the routine-call. Second, it can have *side effects*; that is, it can perform actions other than returning a calculated value, such as modifying data, performing input/output, and so on. The expression "X = F()" calls the routine named F but does not pass any arguments. The value returned by F is assigned to location X.

The following expression calls the routine named P and passes three arguments: the value 5, the contents of location X, and the address of an ASCII string:

```
P(5, .X, UPLIT('MESSAGE'));
```

The value returned by routine P, if any, is not used.

A complete description of routine-calls is given in *Chapter 12, "Routines"*.

4.9. Field-References

A field-reference can designate any portion of storage of up to %BPVAL bits in length. That is, it designates a field value that can range in size from one bit to a fullword. In BLISS–32, for example,

the field can be a sequence of up to 32 bits. Normally, a field-reference is used only within a structure-declaration.

The full description of field-references is given in *Chapter 11, "Data Structures"*.

4.10. Code Comments

A code comment places a comment in the object part of the compilation listing of the module in which it appears. Thus code comments permit annotation of the object code.

In addition, a code comment acts as a barrier to optimizations that are normally performed by the compiler, in that such optimizations do not cross the code comment. Thus it divides the source listing and the object listing into portions that contain mutually corresponding source and object code.

4.10.1. Syntax

```
code comment    CODECOMMENT quoted-string , . . . : block
```

4.10.2. Semantics

The value of a code comment expression is the value of the block.

A code comment places the given quoted-string in the object code listing in the form of an assembly language comment.

A code comment expression prevents code motion. That is, expressions in the source that appear before the code comment expression are compiled into instructions in the object code that precede the generated comment, and source expressions that follow the code comment expression are compiled into instructions that follow the generated comment. A code comment has other effects on optimization. For example, the compiler will not place a value in temporary storage (such as a register) before a code comment and then fetch the value after the code comment. Instead, the compiler recalculates the value.

A general description of optimization is given in the user manual for each BLISS compiler.

Chapter 5. Computational Expressions

The computational expressions of BLISS provide the operations of the language. A single computational expression performs a single basic operation, like addition or the fetching of a value. A combination of computational expressions, nested one within another, can perform a long and complicated sequence of operations.

Computational expressions are classified as either operator-expressions or executable-functions. A typical operator-expression is A=0; it assigns a value, that is, places a value in storage. It is identified by the equal sign (=) operator that appears between the two operands, A and 0. A typical executable-function is MAX(.X,.Y,.Z); it selects the maximum of several values, and it is identified by the keyword MAX that precedes the parameters .X, .Y, and .Z. All computational expressions, regardless of their syntax, perform a predefined operation on given values to produce a result value.

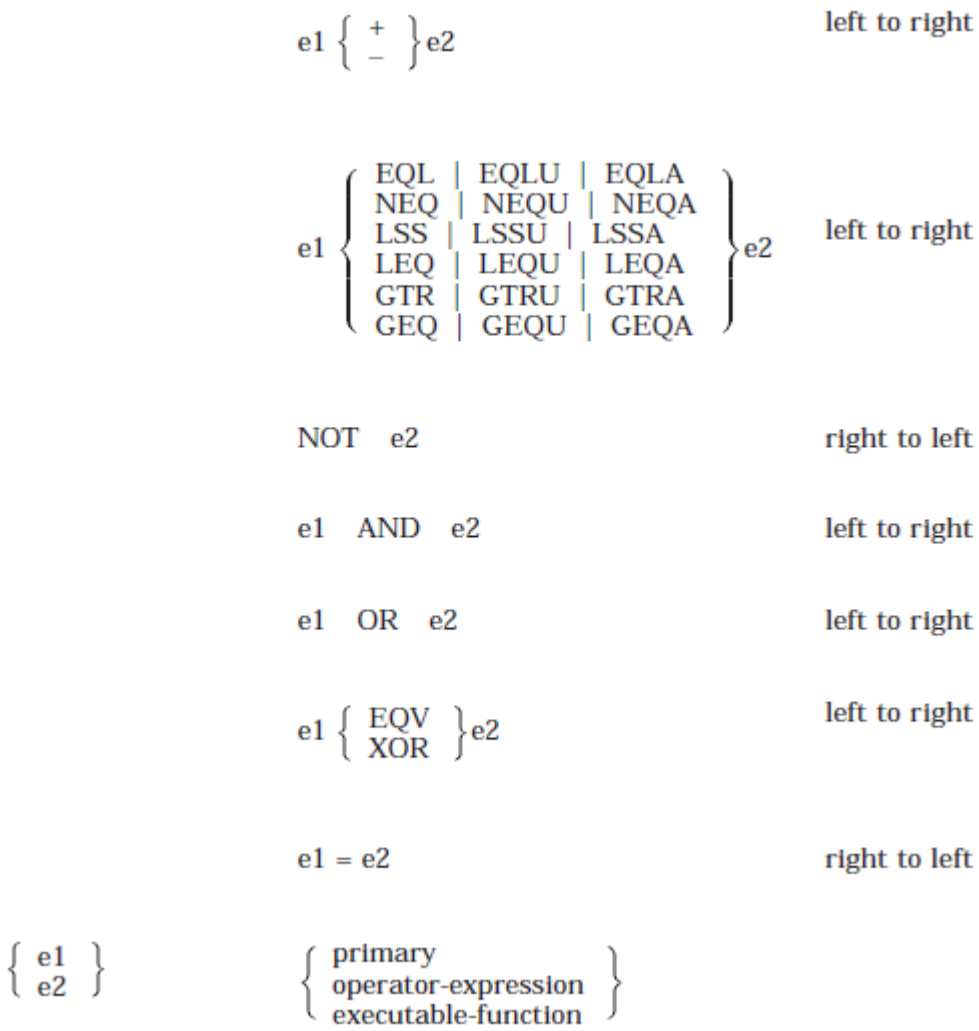
5.1. Operator-Expressions

The notation used for the operator-expressions of BLISS is similar to the notation of mathematics. The terms "operator", "operand", and "associativity" that are used in describing BLISS expressions are all drawn from the terminology of mathematics.

5.1.1. Syntax

The following syntax diagram gives the many forms of the operator-expression. The forms are divided by broken lines into *priority levels*, and an *associativity* is given for each priority level. This information is used in *Section 5.1.3, "Defaults"*.

		Associates from
operator-expression	. e2	right to left
	{ + / − } e2	right to left
	e1 ^ e2	left to right
decreasing priority	e1 { MOD / * / / } e2	left to right



Every operator-expression has one of the following general forms:

```
prefix-operator    right-operand
left-operand       infix-operator     right-operand
```

The operands must be expressions and the operator is either a keyword or a single delimiter character.

5.1.2. Restrictions

An operator-expression must not have an operand that is a control-expression. This restriction is expressed in the syntax (in the rule that defines *e1* and *e2*), but is repeated here for emphasis. For example, the following operator-expression is not valid:

```
X = IF .ALPHA EQL 0 THEN .X1 ELSE .X2
```

Parentheses can be used to avoid this restriction, by converting the right-operand to a compound-expression (see *Section 8.1, "Blocks"* and *Section 5.1.5.1, "Explicit Parenthesization"*).

A *prefix-operator* must not immediately follow an infix or prefix operator that has a higher priority. For example, the following is not valid:

```
.A EQL NOT .B
```

Parentheses can be used to avoid this restriction, by converting the right-operand to a compound-expression.

The result of an arithmetic operation (*, /, MOD, +, and –) must not exceed the capacity of a signed fullword; if it does so, the result is undefined.

The value of the right operand of a modulus (MOD) or division (/) operator must not be zero.

5.1.3. Defaults

The *default parenthesization* for operator-expressions is determined by the priority levels and associativities given in the syntax diagram for operator-expressions. The following rules apply:

1. Parenthesize the operators of a given expression in order of descending priority. That is, first parenthesize all fetch operators (highest priority), then parenthesize all prefix "+" and "–" operators (second highest priority), then continue in this manner through operators of decreasing priority, and finally parenthesize all assignment operators (lowest priority).

2. If an expression contains several occurrences of operators that have a given priority, then parenthesize those operators in the order indicated by the associativity. If the associativity for a given priority level is "left to right", then parenthesize operators with that priority from left to right; if the associativity is "right to left", then parenthesize from right to left.

When an operator is parenthesized, the parentheses surround the operator and the one or two operands required by the operator. For example:

```
3*R(B)-2*.A+12
```

This expression contains four operators, and there are many ways in which it could be explicitly parenthesized. The default parenthesization is obtained as follows:

1. The fetch operator has the highest priority and is parenthesized first, giving the following:

   ```
   3*R(B)-2*(.A)+12
   ```

2. Of the remaining operators in the expression, the two multiplication operators (*) have the highest priority and are parenthesized next, giving the following:

   ```
   (3*R(B))-(2*(.A))+12
   ```

3. The remaining operators (– and +) are used as infix operators. These operators have the same priority level and so associativity must be taken into account. Since associativity is "left to right" for these operators, the subtraction operator (–) is parenthesized first, giving the following:

   ```
   ((3*R(B))-(2*(.A)))+12
   ```

4. Finally, the remaining operator (+) is parenthesized, giving the following:

   ```
   (((3*R(B))-(2*(.A)))+12)
   ```

This fully parenthesized expression is equivalent to the original, unparenthesized expression.

Observe that, in the example just given, the routine-call is treated as a single construct because it is a complete primary. That is, 3*R(B) is parenthesized as (3*R(B)) rather than (3*R)(B). Structure-references and field-references are treated as a single construct in a similar way.

Explicit parenthesization is discussed in *Section 5.1.5.1, "Explicit Parenthesization"*.

5.1.4. Semantics

An operator-expression is evaluated as follows:

1. Evaluate the operand(s) of the expression.

2. Calculate a value according to the specific rules for the given operator.

The value obtained in step 2 is the value of the expression.

In general, the order in which the operands of an operator-expression are evaluated is not defined. See *Section 5.1.5.2, "The Order of Evaluation"*.

The order in which assignment expressions, routine-calls, and control-expressions are evaluated is, however, defined as follows:

Every evaluation of an assignment expression, routine-call, or control-expression in the left operand of an operator-expression is completed before any evaluation of an assignment expression, routine-call, or control-expression in the right operand of the operator-expression is begun. The consequences of this ordering rule are discussed in *Section 5.1.5.2, "The Order of Evaluation"*.

The value of every BLISS expression is a fullword value. It follows that the value of the operands of an operator-expression are fullword values and that the value of the operator-expression itself is a fullword value.

In some cases, an operator-expression produces a value that cannot be represented as a fullword value. In such cases, the value of the expression is undefined and the program is invalid. There is no guarantee that such an overflow is detected or signaled.

The remainder of this description of semantics is devoted to specific rules for the various operator-expressions. The operator expressions are grouped according to function, but they are nevertheless described in the order in which they appear in the syntax diagram, that is, in order of decreasing priority.

5.1.4.1. Fetch Expressions

A fetch expression obtains the value that is stored at a given address. The expression has the following form:

```
. e2
```

The operand of a fetch expression can be a field-reference that has a field-selector; in that case the fetch expression has a special interpretation. However, the use of a field-selector outside of a structure-declaration is not recommended. For that reason, the effect of a field-selector on a fetch expression is described in *Section 11.2, "Field-References"*.

A fetch expression without a field-selector is evaluated as follows:

BLISS–16/32 ONLY

1. If *e2* is the name of a data-segment, then determine its allocation-unit and extension-attribute from its declaration. If *e2* is any other expression, then use the default allocation-unit (WORD for BLISS–16, LONG for BLISS–32) and use UNSIGNED as its extension-attribute.

2. Interpret the value of *e2* as an address. Depending on whether the allocation-unit of *e2* is LONG, WORD, or BYTE, fetch the contents of the longword, word, or byte at that address. LONG and longword apply to BLISS–32 only.

3. If the value fetched in step 2 is a field value (less than %BPVAL bits long), interpret it as a signed or unsigned value depending on the extension-attribute. If the attribute is UNSIGNED, then extend it to a fullword value by placing 0's at the left end. If the attribute is SIGNED, extend it to a fullword value by placing copies of the left-most (sign) bit at the left end.

4. Use the fullword value obtained in step 3 as the value of the fetch expression.

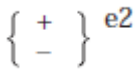
BLISS–36 ONLY

1. Interpret the value of *e2* as an address and fetch the contents of the fullword at that address.

2. Use the fullword value obtained in step 1 as the value of the fetch expression.

5.1.4.2. Prefix Sign Expressions

A prefix sign supplies the algebraic sign for a given value. The expression has the following forms:



The expression is evaluated as follows:

- If the operator is addition (+), then the value of the expression is the value of *e2*.

- If the operator is subtraction (–), then the value of the expression is the negative (two's complement) of the value of *e2*.

5.1.4.3. Shift Expression

This expression performs operations based on the arithmetic shift instruction of the target system. The expression has the following form:

```
e1 ^ e2
```

This operation can be explained in terms of a hypothetical shift register that is valid for all BLISS dialects. The register has *n* bit positions, where *n* is 16, 32, or 36 depending upon the target system (%BPVAL). The positions are numbered starting at the right with position 0 (the low-order position) and ending with position n–1 (the sign position), referred to below as position *m*.

To evaluate an arithmetic shift expression, place the value of *e1* in the shift register and let the value of *e2* be called *v2*. Proceed as follows:

1. If *v2* is positive, move each bit *v2* positions to the left. When a bit is moved out of the sign position, m, discard it. When a bit is moved out of position 0, put a zero-bit in position 0.

2. If *v2* is zero, do not move any bits.

3. If *v2* is negative, move each bit ABS(*v2*) positions to the right. However, do not modify the bit in position m (the sign position). When a bit is moved out of position m–1, put a copy of the sign bit in position m–1. When a bit is moved out of position 0, discard it.

When the shift is complete, use the contents of the shift register as the value of the shift expression.

Sometimes an arithmetic shift is used for scaling; that is, to multiply a value by a power of 2. For that application, the following interpretation of an arithmetic shift is more appropriate:

1. Let *v1* and *v2* be the signed values of the operands and calculate the following value:

   ```
   v1 * (2**v2)
   ```

   In this expression, 2**v2 means "2 to the power v2".

2. If the result of step 1 is not an integer, reduce it to the next smallest integer. For example, reduce 2.5 to 2 and reduce –2.5 to –3.

3. Represent the result of step 2 as a signed, two's complement binary integer. If the result requires more than %BPVAL bits for its representation, some of the high-order bits of the representation are lost.

This interpretation is entirely equivalent to the interpretation in terms of a shift register; it is just another way of looking at the same operator.

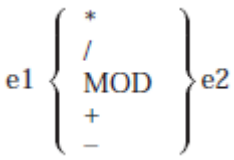
Examples of arithmetic shift operations are given in the following table:

v1	**v2**	**2**v2**	**v1*(2**v2)**	**v1^v2**
10	2	4	40	40
–10	2	4	–40	–40
10	–2	0.25	2.5	2
–10	–2	0.25	–2.5	–3

All the values in this table are decimal numbers. Observe that when *v2* is positive, the arithmetic shift performs multiplication by a power of 2. When *v2* is negative and *v1* is positive, the shift performs division by a power of 2. When *v2* and *v1* are both negative, the shift performs something close to, but not quite the same as, division by a power of 2.

5.1.4.4. Arithmetic Expressions

The multiplication, division, addition, and subtraction expressions perform the operations of ordinary arithmetic. The modulus (MOD) expression obtains the remainder of a division. The expression has the following form:



The values of the operands are interpreted as signed values, and the result is represented as a signed value. If the result is outside the range provided by a signed fullword, then the expression is invalid and the value of the expression is undefined.

Let *v1* and *v2* be the values of the operands. The expression is evaluated as follows:

- If the multiplication operator (*) is used, then multiply *v1* by *v2* and use the result as the value of the expression.
- If the division operator (/) is used, then proceed as follows:
  - If *v2* is zero, the expression is invalid and the value of the expression is undefined.
  - Otherwise, divide *v1* by *v2*. If the result is not an integer, drop its fractional part without rounding (so that 2.8 becomes 2 and –2.8 becomes –2). Use the result as the value of the expression.
- If the modulus operator (MOD) is used, then proceed as follows:
  1. If *v2* is zero, the expression is invalid and the value of the expression is undefined.
  2. Otherwise, divide *v1* by *v2*. Drop the fractional part of the value (so that 2.8 becomes 2.0 and –2.8 becomes –2.0).
  3. Multiply the value obtained in step b by *v2*.
  4. Subtract the value obtained in step c from *v1* and use the result as the value of the expression.
- If the addition operator (+) is used, then add *v2* to *v1* and use the result as the value of the expression.
- If the subtraction operator (–) is used, then subtract *v2* from *v1* and use the result as the value of the expression.

The MOD operator is the remainder of the division of *v1* by *v2*. An aid to understanding the MOD operator is the following identity:

```
(v1 MOD v2) EQL (v1-v2*(v1/v2))
```

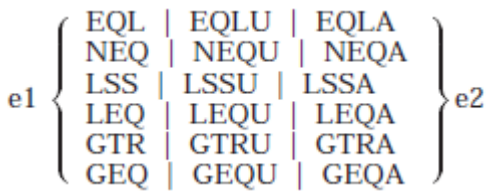
Some examples of the division (/) and modulus (MOD) operations follow:

v1	**v2**	**v1/v2**	**v1 MOD v2**
10	3	3	1
10	–3	–3	1
–19	7	–2	–5
–19	–7	2	–5
13	2	6	1
13	8	1	5
13	10	1	3
13	16	0	13

The last four examples show how the MOD operator is used to obtain the last digit of the binary, octal, decimal, and hexadecimal representations of 13.

5.1.4.5. Relational Expressions

A relational expression is used to compare two values. The expression has the following form:



The interpretation of the operator itself is determined by the first three letters of the operator, as follows:

EQL	is equal to
NEQ	is not equal to
LSS	is less than
LEQ	is less than or equal to
GTR	is greater than
GEQ	is greater than or equal to

The interpretation of the operands is determined by the fourth letter of the operator as follows:

No fourth letter:	Interpret operand values as signed values.
Fourth letter is U:	Interpret operand values as unsigned values.
Fourth letter is A:	Interpret operand values as address values.

If the values of the operand satisfy the relation specified by the operator, then the value of the relational expression is "1"; otherwise, it is "0". In both cases, the value is represented as a fullword value.

In both BLISS–16 and BLISS–32, the operators LSSU and LSSA are equivalent, as are GTRU and GTRA, LEQU and LEQA, and GEQU and GEQA. That is, the unsigned and address forms of the magnitude sensitive relational operators are equivalent. In BLISS–36, however, the operators LSS (signed) and LSSA are equivalent, as are GTR and GTRA, and so on. This reflects a difference in the range of valid address values allowed by the corresponding systems. The distinction between the signed/unsigned and the address forms of the operators is provided so that programmers can specify the desired interpretation of the values being operated on, in both an explicit and a transportable fashion.

Note that all forms of the EQL and NEQ operators are by nature equivalent in all dialects; the unsigned and address forms are provided for symmetry with the other relational operators discussed above. Use of the alternate forms is encouraged for the sake of clarity.

Two examples of the use of relational expressions follow:

Expression	**Value**
-1 LSS 0	1 (true)
-1 LSSU 0	0 (false)

As another example, consider the following program fragment:

```
OWN
    X,
    Y;
...
```

```
X LSSA Y
```

The value of the relational-expression in this example is 1 (true) because X is allocated at a smaller address than Y.

5.1.4.6. Boolean Expressions

A Boolean expression is used to apply a Boolean operation to given values. The expression has the following forms:

NOT e2

e1 { AND / OR / XOR / EQV } e2

Each of these expressions operates on the individual bits of the operands to produce the individual bits of the result. The specific rules are as follows:

- If the operator is NOT, then the ith bit of the result is obtained from the ith bit of the value of *e2* according to the following table:

e2	**NOT**
0	1
1	0

- If the expression has two operands, then the ith bit of the result is obtained from the ith bit of the value of *e1* and the ith bit of the value of *e2* according to the following table:

e1	**e2**	**AND**	**OR**	**XOR**	**EQV**
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	1

The appropriate rule is applied %BPVAL times, once for each bit in the result.

Boolean logic applies to single bits while BLISS always operates on fullwords. Therefore special precautions are sometimes required in programming Boolean logic in BLISS.

For example, if A is the name of a Boolean variable whose value is always 0 or 1, and the negation of the contents of A must be assigned to another Boolean variable named B, you might try the following assignment:

```
B = (NOT .A);
```

However, this assignment does not produce a Boolean value. Instead, its effect (assuming a BLISS–32 fullword) is as follows:

Contents of A	Contents of B
0	11111111111111111111111111111111 (binary)
1	11111111111111111111111111111110 (binary)

The low-order bit is the desired Boolean result, but the other bits clutter up the result. To assign a Boolean value to B, the high-order bits can be masked out as follows:

```
B = ((NOT .A) AND 1); or B = .A XOR 1;
```

5.1.4.7. Assignment Expressions

An assignment expression is used to store a given value at a given address. The form of the expression is as follows:

```
e1 = e2
```

The left operand of an assignment expression can be a field-reference that has a field-selector; in that case the assignment expression has a special interpretation. However, the use of a field-selector is not recommended outside of a structure-declaration. For that reason, the effect of a field-selector on an assignment expression is described later, in *Section 11.2, "Field-References"*. An assignment-expression without a field-selector is evaluated as follows:

BLISS–16/32 ONLY

1. If *e1* is the name of a data segment, then determine its allocation-unit from its declaration. If *e1* is any other expression, then use the default allocation-unit (WORD for BLISS–16, LONG for BLISS–32).

2. Interpret the value of *e1* as an address. Depending on whether the allocation-unit of *e1* is LONG, WORD, or BYTE, store the corresponding number of rightmost bits of the value of *e2* in the longword, word, or byte at the given address. LONG and longword apply to BLISS–32 only.

3. Use the original value of *e2* (that is, the fullword value) as the value of the assignment expression.

BLISS–36 ONLY

1. Interpret the value of *e1* as an address and store the value of *e2* in the fullword at the given address.

2. Use the value of *e2* as the value of the assignment expression.

5.1.5. Pragmatics

Two aspects of the interpretation of operator-expressions are discussed here: the effect of explicit parenthesization, and the order of expression evaluation.

5.1.5.1. Explicit Parenthesization

Any expression can be placed in parentheses. The value of the parenthesized expression is the value of the expression within the parentheses. The effect of the parentheses is to delimit the operands of the expression. Consider the following expressions:

```
(.A)+1

.(A+1)
```

The two different placements of the parentheses produce two expressions that are not equivalent. In the first example, the operand of the fetch operator is just A, while in the second example, it is A+1.

Every expression is fully parenthesized, if necessary, by the compiler to determine which operands go with each operator, according to the default rules given in *Section 5.1.3, "Defaults"*. For example, the default parenthesization of the expression .A+1 is as follows:

```
(.A)+1
```

This parenthesization follows from the fact that the fetch operator has higher priority than the addition operator. The expression could be explicitly parenthesized as follows, however, to specify the interpretation required:

```
.(A+1)
```

Sometimes an operator-expression must be explicitly parenthesized because of restrictions that prohibit the use of certain operands (see *Section 5.1.2, "Restrictions"*). Any operand can, itself, be a parenthesized expression because a parenthesized expression is a form of block (as defined in *Section 8.1, "Blocks"*), which is a primary (as defined in *Section 4.1, "Primaries"*). For example, the following expression is valid but the unparenthesized form is not:

```
X = (IF .ALPHA EQL 0 THEN .X1 ELSE .X2)
```

Again, the following expression is valid, but the unparenthesized form is not:

```
.A EQL (NOT .B)
```

5.1.5.2. The Order of Evaluation

As stated in *Section 5.1.4, "Semantics"*, the order in which operator-expressions are evaluated is largely undefined. By leaving the order undefined, the language definition permits the compiler to choose an order of evaluation that is efficient.

In most cases, the results of programs are not affected by the absence of a defined order of evaluation. For example:

```
X = 2*.X + 3/.Y;
```

The absence of a defined order of evaluation does not affect the value assigned to X because all possible orders of evaluation of this assignment (after the operands are delimited by default parenthesization) produce the same value.

The rule near the beginning of *Section 5.1.4, "Semantics"*, however, states that assignment expressions, routine-calls, and control-expressions are evaluated in left-to-right order. In some cases where the order of evaluation is important, this rule provides the necessary ordering. For example:

```
BETA = 2*R(.Y) + Q(.Z)
```

If R and Q are names of routines, and the routines they designate use the same data (for example, R sets a data segment that Q fetches), then it is important that the routines be called in the indicated order.

It must be said, however, that the example just given is not good programming. It is legitimate for a routine-call to set or use data that is not mentioned in the routine-call, but a dependence between two routine-calls in the same expression is dangerously obscure.

Some expressions are invalid because they depend on an ordering that is undefined. For example:

```
Q = .X + (X=.Y);
```

It is not valid to assume that the contents of X will be fetched before it is set. The value assigned to Q could be either the value of .X+.Y or the value of 2*.Y. Assuming that it was the first of the two values that was intended, you can revise the example by breaking it into two assignments, as follows:

```
Q = .X + .Y;
X = .Y;
```

This version is valid because expressions that are separated by a semicolon are always evaluated in sequence, one at a time.

The example just given was quite obviously bad programming. However, the same problem can arise with certain routine-calls, and then the problem is less obvious. As an example, suppose that routine R contains, among other things, the following assignment expression:

```
X = .Y;
```

Now consider the following expression:

```
Q = .X + R();
```

This statement has the same problem as the earlier one; there is no rule that specifies whether the operator that fetches X or the call on the routine R is evaluated first.

5.1.5.3. Operations on Field Values in BLISS–16/32

When all data segments involved in a calculation occupy fullwords, the calculation is relatively easy to program. Fullwords accommodate large values, and assignment from one fullword to another never modifies a value.

When a data segment that is smaller than a fullword is involved in a calculation, problems can arise, either through the assignment of a large value to the small data segment or through the incorrect extension of the contents of the small data segment. An example of the latter problem follows:

```
OWN
    X: BYTE,
    Y;
...
X = -1;
Y = .X + 1
```

For purposes of discussion, assume that there is a good reason for restricting X to one byte. Because X does not occupy a fullword, it is extended before being incremented and assigned to Y. And because X is UNSIGNED by default, the extended value is 255 rather than –1. Thus the value of Y becomes 256 rather than 0.

The program fragment under discussion does not violate any rules of BLISS–16 or BLISS–32; it is valid. However, since it assigns a negative number, –1, to a name that is declared UNSIGNED by default, the program fragment is certainly inconsistent.

You can fix the program in either of the following ways:

- Change the numeric literal from –1 to 255. This change does not affect the value assigned to Y, but it does make it clear that you expect that result.
- Insert the SIGNED attribute to the declaration of X. This change causes 0 to be assigned to Y.

The choice between these changes depends entirely on your intentions and cannot be made by looking at this small part of the program. Related problems can arise (in any dialect) from the use of field-references for fields that are smaller than a fullword. These are discussed in *Section 11.2.5.4, "Operations on Scalar Field Values"*.

5.2. Executable-Functions

The executable-functions are called "executable" to distinguish them from the lexical-functions, which are described in *Chapter 15, "Lexical Functions"*. There are five kinds of executable-functions:

standard-functions
supplementary-functions
condition-handling-functions (BLISS–16/32 only)
linkage-functions
machine-specific-functions

Each of these kinds of function is described in the following paragraphs.

The *standard-functions* are general-purpose functions; that is, they are restricted to neither a specific area of system programming nor a specific computer system. The standard-functions are just as fundamental to BLISS as the operator-expressions. An example of a call on a standard-function follows:

```
MAX(.X, .Y, 0)
```

The value of this function is the contents of X, the contents of Y, or 0, whichever is greatest. The name MAX is predeclared as an executable-function, so the example just given can appear where MAX is undeclared. The standard-functions are defined in this chapter (*Section 5.2.2, "Semantics"*).

The *supplementary-functions* are designed for particular areas of system programming. These functions are usually defined and documented in "packages". One such package consists of the character-handling functions. An example of a call on such a function follows:

```
X = CH$RCHAR(.PTR3);
```

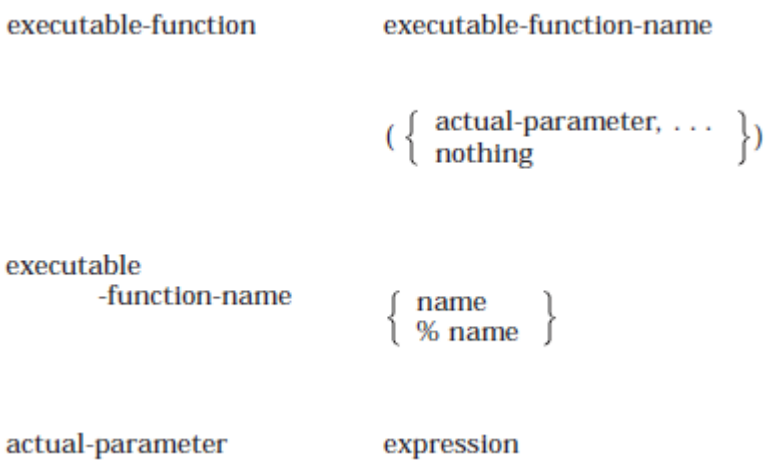
This assignment reads a character from the position selected by the contents of PTR3 and assigns it to X. The character-handling functions are the only supplementary-functions defined in this manual. However, it is anticipated that other packages of supplementary-functions will be added to the language in the future.

The *condition-handling-functions* are used for generating signals for unusual events or conditions and for controlling the subsequent processing of a signal (BLISS–16/32 only). These functions are defined in *Chapter 17, "Condition Handling"*.

The *linkage-functions* are used in combination with some linkages (calling sequences) to code routines in a more general way; for example, to code a routine that can be called with different numbers of parameters in different calls. The linkage-functions are defined in *Section 13.6, "Linkage-Functions"*.

The *machine-specific-functions* are designed for specific computer systems. Usually a machine-specific-function represents a single hardware instruction. Such a function permits the use of a hardware instruction without digressing to an assembly language. The use of a machine-specific-function makes a program machine dependent. An example of the use of a machine-specific-function is not given here. Such an example would be misleading without a detailed description of the context in which it appeared. The use of machine-specific-functions requires knowledge of both the hardware instruction set and the optimization strategies of the compiler. Machine-specific-functions are described in the respective BLISS user manual.

5.2.1. Syntax



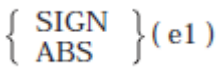
5.2.2. Semantics

The semantics of the executable-functions is nearly identical to that for operator-expressions (see *Section 5.1, "Operator-Expressions"*). The only difference is that the operation to be performed is specified by a name at the beginning of the executable-function (for example, MAX) instead of by an operator.

The semantics of the standard-functions are given in the following subsections. The semantics of some supplementary-functions, the character-handling functions, are given in *Chapter 20, "Character-Handling Functions"*. The semantics of the machine-specific-functions are defined in the user manual for each dialect.

5.2.2.1. SIGN and ABS Functions

The SIGN and ABS functions are used to extract the sign and the absolute value, respectively, from a value. The functions have the following form:



Either of these functions is a compile-time-constant-expression if its actual-parameter is a compile-time-constant-expression. The values returned by these functions are as follows:

Function	**Value**	
SIGN(x)	+1	if x > 0
	0	if x = 0
	–1	if x < 0
ABS(x)	x	if x 0
	–(x)	if x < 0

Examples of the use of the SIGN and ABS functions are as follows:

Example	**Value**

SIGN(5)	+1
ABS(5)	+5
SIGN(–5)	-1
ABS(–5)	+5
SIGN(0)	0
ABS(0)	0

Observe that, in each of these examples, the following expression is true:

```
SIGN(x)*ABS(x) EQL x
```

5.2.2.2. MAX and MIN Functions

The MAX and MIN functions are used to select the largest and the smallest, respectively, from a set of values. The functions have the following form:

```
{ MAX | MAXU | MAXA }
{ MIN | MINU | MINA }(e1 , e2 , . . . )
```

The interpretation of the function itself is determined by the first three letters of its name, as follows:

MAX	select the largest value
MIN	select the smallest value

The interpretation of the operands is determined by the fourth letter of the function name as follows:

No fourth letter:	Interpret operand values as signed values.
Fourth letter is U:	Interpret operand values as unsigned values.
Fourth letter is A:	Interpret operand values as address values.

The value of the function is the largest or smallest of the values of the operands, depending on the function name.

In both BLISS–16 and BLISS–32, the functions MAXU and MAXA are equivalent, as are MINU and MINA. That is, the unsigned and address forms of the MAX and MIN functions are equivalent. In BLISS–36, however, the functions MAX (signed) and MAXA are equivalent, as are MIN and MINA. This reflects a difference in the range of valid address values allowed by the corresponding systems.

The distinction between the signed/unsigned and the address forms of the functions is provided so that programmers can specify the desired interpretation of the values being operated on, in a both explicit and transportable fashion.

Examples of the use of the signed and unsigned maximum and minimum functions are as follows:

Example	**Value**
MAX(–1,0,1)	1
MAXU(–1,0,1)	-1

MIN(–1,0,1)	-1
MINU(-1,0,1)	0

These examples show the difference between the signed and unsigned functions. The signed functions treat –1 (which is represented as a fullword of 1s) as a negative value, whereas the unsigned functions treat –1 as a large positive value.

An example of the use of the address maximum and minimum functions is as follows:

```
OWN
    X: VECTOR[10],
    Y,
    Z;
...
Z = MAXA(X[5],Y)
```

The assignment sets Z to the value of Y because OWN data segments are allocated at increasing addresses.

5.2.2.3. The %REF Function

The %REF function provides temporary storage for the value of an actual-parameter in a routine-call or executable-function. The function has the following form:

```
%REF ( e1 )
```

The function can be used only as an actual-parameter in a routine-call or executable-function.

The function is evaluated as follows:

1. Allocate a temporary fullword and place the value of *e1* in that fullword.

2. Use the address of the temporary fullword as the value of the function.

For purposes of discussion, suppose that a programmer has declared a routine called RHO. The details of the declaration are not given here. All that matters is that the routine has one parameter, which is the address of a given value, and returns a result which, presumably, depends on the given value.

Suppose, now, that the value to be passed is not stored in a data segment but must, instead, be calculated. Specifically, it is the value of the expression: .X+1. It would not be correct to write the following:

```
Y = RHO(.X+1);
```

In this version, .X+1 would not be used as the given value (which was intended), but rather as the address of the given value. A correct solution to the problem is to declare and use a temporary data segment name. However, the use of a temporary just to deal with a calculated parameter is inconvenient. The %REF function provides a better solution, as follows:

```
OWN
    X,
    Y;
...
Y = RHO(%REF(.X+1));
```

Observe that %REF is not an "undot" operation. The following calls are not equivalent:

```
F(X)
```

```
F(%REF(.X))
```

The routine-call F(X) passes the address of X as the actual-parameter of the routine F, while the second call passes the address of a temporary data segment that contains a copy of the contents of X.

5.2.3. Pragmatics

The cost of evaluating a typical executable function is much less than the cost of evaluating a typical routine-call. The use of an executable-function usually does not produce a routine call; instead, it is compiled into a few instructions that are often designed precisely for the required operation. In contrast, a routine-call usually requires the passing of parameters, the creation of a stack frame, and the return of a result as well as the inevitable subroutine jump. In fact, the similarity between an executable-function and a routine-call does not extend much beyond the similarities in their syntax.

Chapter 6. Control Expressions

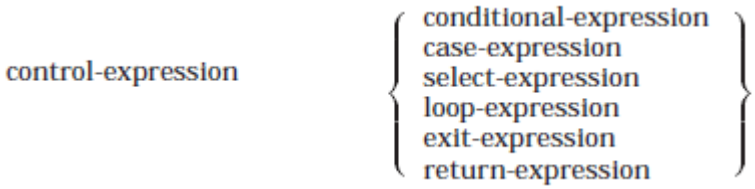
Early programming languages permitted unrestricted patterns of control flow, and the logic of many programs was very difficult to follow. More recent languages have introduced specialized and restricted patterns of flow, and thus encourage the construction of programs that are better organized.

There are five fundamental kinds of control flow in BLISS: sequential, conditional, iterative, subroutine, and condition handling. Sequential flow, a simple notion, is defined in *Section 8.1.3, "Semantics"* as part of the description of blocks. Conditional and iterative flow is described in this chapter. Subroutine flow is described in *Chapter 12, "Routines"*, and condition handling in *Chapter 17, "Condition Handling"*.

Notable by its absence in BLISS is the familiar GO TO construct. Its absence prevents the use of arbitrary patterns of flow. Programming without the GO TO frequently requires more analysis of the problem, but usually results in a clearer and more reliable program.

In BLISS, the constructs for conditional and iterative flow control are called *control-expressions*. Because they are expressions, these constructs can have values and can be nested within larger expressions.

The syntax diagram for control-expressions is as follows:



Loop-expressions are described under two categories: indexed-loops and tested loops.

6.1. Conditional-Expressions

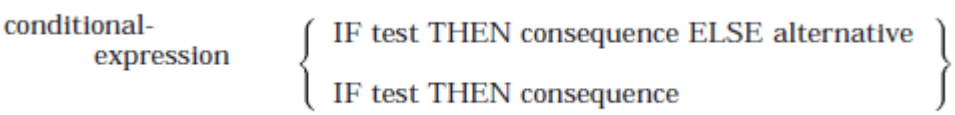
A conditional-expression performs a given test and then, depending on whether or not the test is satisfied, evaluates the first or second of two given expressions.

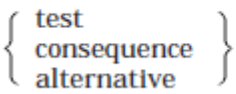
An example of a conditional-expression follows:

```
IF .X GTR XMAX THEN F(.X) ELSE G(.X);
```

In this example the contents of X is compared with a value XMAX. If .X is greater than XMAX, then the routine F is called; otherwise, routine G is called.

6.1.1. Syntax





expression

In addition to these syntactic rules, the following rule also is required:

```
An ELSE alternative always modifies the closest IF-THEN in a
 conditionalexpression.
```

An example of an expression to which this rule applies follows:

```
IF .A EQL 0 THEN IF .B EQL 0 THEN X = 5 ELSE X = 6;
```

This expression is interpreted as follows:

```
IF .A EQL 0 THEN (IF .B EQL 0 THEN X = 5 ELSE X = 6);
```

6.1.2. Restrictions

A conditional-expression that lacks an "ELSE alternative" must not be used in a context that requires a value.

6.1.3. Semantics

The *satisfaction of a test* depends on the low-order (rightmost) bit of the value of the test. If the low-order bit is 1, the test is satisfied; otherwise, the test is not satisfied.

Expressions used as test expressions are subject to an evaluation rule that is more flexible (for optimization purposes) than the rule applied in other contexts. Specifically, the test-expression evaluation rule is as follows:

```
Within a test expression, an expression that is not needed to determine the
 value of the test expression is not necessarily evaluated.
```

A test expression that is subject to this rule appears in the following conditional-expression:

```
IF .A OR F(.B) THEN X = 0
```

If the contents of A is 1 (true), then the value of the entire test expression is 1 (true) regardless of the value of F(.B). Consequently, the call on routine F may not be evaluated. Writing the test in the reverse order does not change the situation (see *Section 6.1.4.3, "Complete Versus Incomplete Test Evaluation"*).

Given the preceding description of test evaluation, the interpretation of an entire conditional-expression is as follows:

1. Evaluate the test.

2. If the test is satisfied, evaluate the consequence and use that value as the value of the conditional-expression.

3. If the test is not satisfied and if an alternative is present, evaluate the alternative and use that value as the value of the conditional-expression. If an alternative is not present, the value of the expression is undefined.

6.1.4. Pragmatics

The following concerns the nesting of conditional expressions, the use of returned values, and proper test evaluation.

6.1.4.1. Nesting of Conditional Expressions

Conditional expressions provide a way to choose one of two mutually exclusive actions, depending on a specified test condition. The *test*, *consequence*, or *alternative* can be any expression. It is common, for example, for the consequence or alternative to be a sequence of expressions (written as a block) as in the following:

```
IF .X EQL 0
THEN (Y = .Y+1; F(.Y); G())
ELSE (G(); Y = .Y-1);
```

Control expressions can also be included in these expressions. For example:

```
IF (IF .X EQL 0 THEN .Y ELSE F(.Y))
THEN
    Z = G() + 5;
```

In this example, the following conditional-expression appears as the test expression of another, larger conditional-expression:

```
IF .X EQL 0 THEN .X ELSE F(.Y)
```

The inner test (.X EQL 0) determines which of the two expressions (.Y or F(.Y)) is used as the test for the outer conditional.

6.1.4.2. Used Versus Discarded Values

Every BLISS expression has a value; however, in some contexts that value is *used* and in others it is *discarded*. This aspect of BLISS is discussed here because the conditional-expression is a good example of an expression that works as well in both contexts. However, the following discussion applies to the value of any kind of BLISS expression.

An example of a conditional-expression whose value is *used* is as follows:

```
D = (IF .I EQL .J THEN 20 ELSE 30);
```

If .I and .J are equal, then 20 (which is the value of the consequence) becomes the value of the conditional-expression and is assigned to D. Note that, because the assignment expression is followed by a semicolon, its value is discarded, but only after the assignment has been performed.

An example of a conditional-expression whose value is *discarded* is as follows:

```
IF .I EQL .J THEN D = 20 ELSE D = 30;
```

Assume that .I and .J are again equal; then the evaluation of the consequence causes 20 to be assigned to D and also causes 20 to be the value of the conditional-expression. Since the conditional-expression is followed by a semicolon, its value is discarded.

The two expressions just given are equivalent in function, and are close enough in their cost that the choice between the two examples is ordinarily a matter of programming style.

6.1.4.3. Complete Versus Incomplete Test Evaluation

As *Section 6.1.3, "Semantics"*stated, a test may not be fully evaluated. Furthermore, different occurrences of the same test may be evaluated in different ways. These variations reflect the fact that the BLISS

compiler performs a far-reaching analysis of the context in which a test appears and then produces code that is optimized for that context. For this reason, an expression that must be evaluated (because it sets values or has other side effects) must not be part of a test.

If an assignment or routine-call must be evaluated, its value should be assigned to a temporary variable. Then the value of the temporary variable can be used in the test expression. For example:

```
IF .A OR F(.B) THEN X = 0;
```

This can be rewritten as follows:

```
T = F(.B);
IF .A OR .T THEN X = 0;
```

6.2. Case-Expressions

A case-expression evaluates an index and then uses the value of that index to choose one expression to be evaluated from a set of expressions.

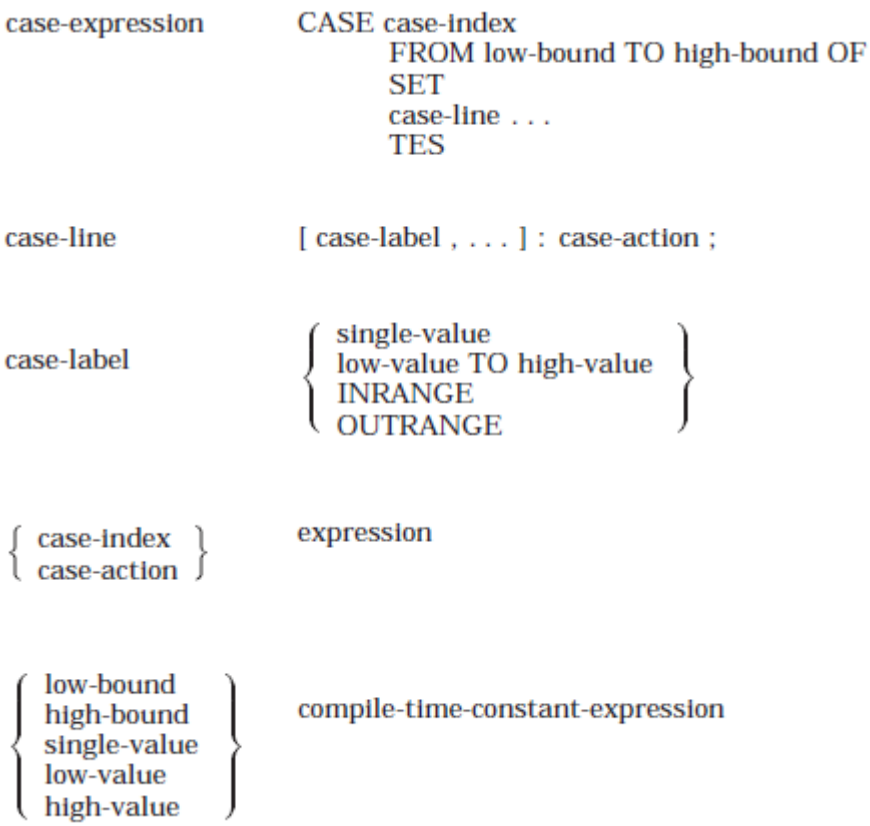
An example of a case-expression is as follows:

```
CASE .X+1 FROM -1 TO 8 OF
    SET
    [1]: F1();
    [2 TO 4]: F2();
    [5, 7, -1]: F3();
    [INRANGE]: F4();
    [OUTRANGE]: F5();
    STES
```

In this example, the value of .X+1 is used to choose one of five routines to be called as follows:

Value of .X+1	**Routine Called**
–1	F3
0	F4
1	F1
2	F2
3	F2
4	F2
5	F3
6	F4
7	F3
8	F4
(all other values)	F5

6.2.1. Syntax



6.2.2. Restrictions

Every value within the range specified by the low-bound and high-bound expressions must be accounted for exactly once in a case-expression. If an integer value in the range is not explicitly given, a case-action must be specified for INRANGE.

If the case-index can assume a value outside the specified range, a case-action must be specified for OUTRANGE.

If the INRANGE case-label is used, it must appear after all case-labels of one of the following forms:

```
single-value
```

```
low-value TO high-value
```

Thus, the only case-label that can follow INRANGE is OUTRANGE.

6.2.3. Semantics

The *matching* of the case-index to a case-label determines the case-action to be evaluated. The syntax provides four kinds of case-label. The following list gives, for each kind of case-label, the condition under which a match occurs.

Case-Label	Condition for a Match

single-value	A match occurs if the values of the case-index and the single-value are equal.
low-value TO high-value	A match occurs if the value of the case-index is in the range specified by the values of the low-value and high-value expressions (that is, the following signed comparisons hold: low-value case-index high-value).
INRANGE	A match occurs if the value of the case-index is in the range specified by the values of the low-bound and high-bound expressions (that is, the following signed comparisons hold: low-bound case-index high-bound) and the case-index does not match any other case-label.
OUTRANGE	A match occurs if the value of the case-index is outside the range specified by the values of the low-bound and high-bound expressions.

Given the preceding definition of matching, the interpretation of an entire case-expression is as follows:

1. Evaluate the case-index.

2. Evaluate the case-action in the case-line that contains the case-label matched by the case-index.

3. Use the value of the case-action as the value of the case-expression.

The case-expression is designed for a special, very efficient implementation. In order to make a decision about using a case-expression, you need to understand its implementation. Briefly, the bounds and case-labels of a case-expression are all compile-time-constant-expressions and can therefore be evaluated by the compiler. For this reason, the compiler can prepare a *transfer vector* for use in the evaluation of a case-expression. The transfer vector has one element for each value of the case-index in the range from low-bound to high-bound. The first element of the vector provides the address of the object code for the case-action that is performed when the case-index is equal to low-bound. The second element provides the address of the object code for the case-action that is performed when the case-index is equal to low-bound plus 1; and so on.

When a case-expression is evaluated during program execution, only a single operation is required to get to the appropriate case-action. That is, the case-index is used as an index into the transfer vector. Thus a case-expression does not require a search through the case-labels.

6.2.4. Pragmatics

A case-expression is most useful when the case-index assumes values in a small range. An example of the effective use of a case-expression follows:

```
CASE .TYPECODE FROM 0 TO 3 OF
    SET
    [0]: LITERAL();
    [1]: IDENTIFIER();
    [2]: KEYWORD();
    [3]: PREDCL();
    TES;
```

This case-expression is used to choose the routine to be evaluated based on the value of .TYPECODE. The data segment named TYPECODE contains a code that is set earlier in the program. Since TYPECODE cannot assume a value outside the specified range, a case-action is not given for OUTRANGE and since each of the values within the range is associated with a specific case-action, a case-action is not given for INRANGE.

Another example of a case-expression follows:

```
CASE .NUMBER FROM 1 TO 10 OF
    SET
    [1,2,3,5,7]: PRIME = .PRIME + 1;
    [INRANGE]: NONPRIME = .NONPRIME + 1;
    [OUTRANGE]: ERROR();
    TES;
```

This case-expression increments the counter PRIME if the contents of NUMBER is 1, 2, 3, 5, or 7. If the contents of NUMBER is 4, 6, 8, 9, or 10, the counter NONPRIME is incremented. If the contents of NUMBER is outside the specified range, an error routine is called.

6.3. Select-Expressions

A select-expression evaluates an index and then uses the value of that index to choose one or more expressions to be evaluated. Two kinds of select-expressions are defined for BLISS: one evaluates all expressions chosen by the index, and the other only evaluates the first such expression. A select-expression differs from a case-expression in several important ways:

- Select-labels are evaluated at execution time.
- A range of values is not specified for the select-index.
- The select-index and select-labels can be interpreted as signed, unsigned, or address values depending on the form of the select expression used.

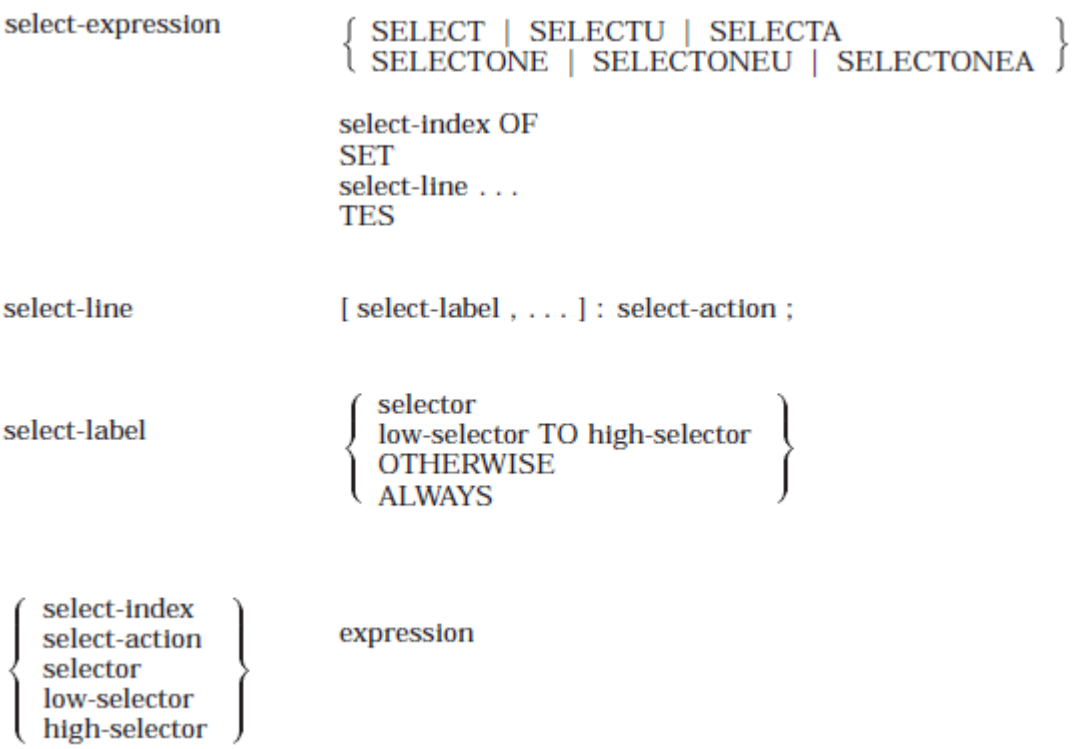
The following select-expression example assumes the VAX–11/780 target system:

```
SIZE=(SELECTONE .VALUE OF
    SET
    [-128 TO 127]:       1;
    [-32768 TO 32767]:   2;
    [OTHERWISE]:         4;
    TES);
```

In this example, the contents of VALUE is used to determine the number of bytes of storage needed for its representation.

If the select-expression in this example is reprogrammed as a case-expression, it requires a range from –32768 to 32767, and its transfer vector occupies 65536 16-bit words. For this reason, the case-expression is decidedly impractical for this example. The particular example used and the transfer-vector size cited are not appropriate for all target systems, of course, but do convey the essential differences between select- and case-expression usage.

6.3.1. Syntax



6.3.2. Restrictions

The select-label ALWAYS cannot be used in an expression that begins with SELECTONE, SELECTONEU, or SELECTONEA.

6.3.3. Semantics

The *matching* of the select-index to a select-label determines whether or not the select-action in the select-line containing the select-label is evaluated. The syntax provides four kinds of select-label. The following list gives, for each kind of select-label, the condition under which a match occurs.

Select-Label	Condition for a Match
selector	A match occurs if the values of the select-index and selector are equal.
low-selector TO high-selector	A match occurs if the value of the select-index is in the range specified by the values of the low-selector and high-selector expressions (that is, low-selector select-index high-selector).
OTHERWISE	A match occurs if a match has not previously occurred.
ALWAYS	A match always occurs.

The keyword at the beginning of a select-expression consists of SELECT or SELECTONE, followed by an optional added letter, U or A. The added letter affects the matching of the select-index to a particular select-label. Specifically, it determines the kind of comparison, as follows:

No added letter:	Use signed comparison.
Last letter is U:	Use unsigned comparison.
Last letter is A:	Use address comparison.

Given the preceding discussion of matching and keywords, the interpretation for an entire select-expression is as follows:

1. Evaluate the select-index.
2. Let the first select-line of the select-expression be the *current* select-line.
3. Evaluate the select-labels on the current select-line to determine whether at least one of them matches the select-index.
4. If a match is found, then evaluate the select-action of the current select-line. Otherwise, go to step 6.
5. If the select-expression is a form of SELECTONE, then go to step 8.
6. If the current select-line is the last select-line, then go to step 8.
7. Let the select-line that follows the current select-line be the new current select-line and go to step 3.
8. Use the value of the most recently evaluated select-action as the value of the select-expression. If no select-action has been evaluated during this evaluation of the select-expression, use –1 as the value of the select-expression.

In step 3 of this interpretation, the select-labels in a single select-line may be evaluated in any order. Furthermore, they are subject to partial evaluation in the same way as a test in a conditional-expression (see *Section 6.1.3, "Semantics"*). Therefore, a select-label must not contain assignments or routine-calls that must be evaluated because they have important side effects.

6.4. Indexed-Loop-Expressions

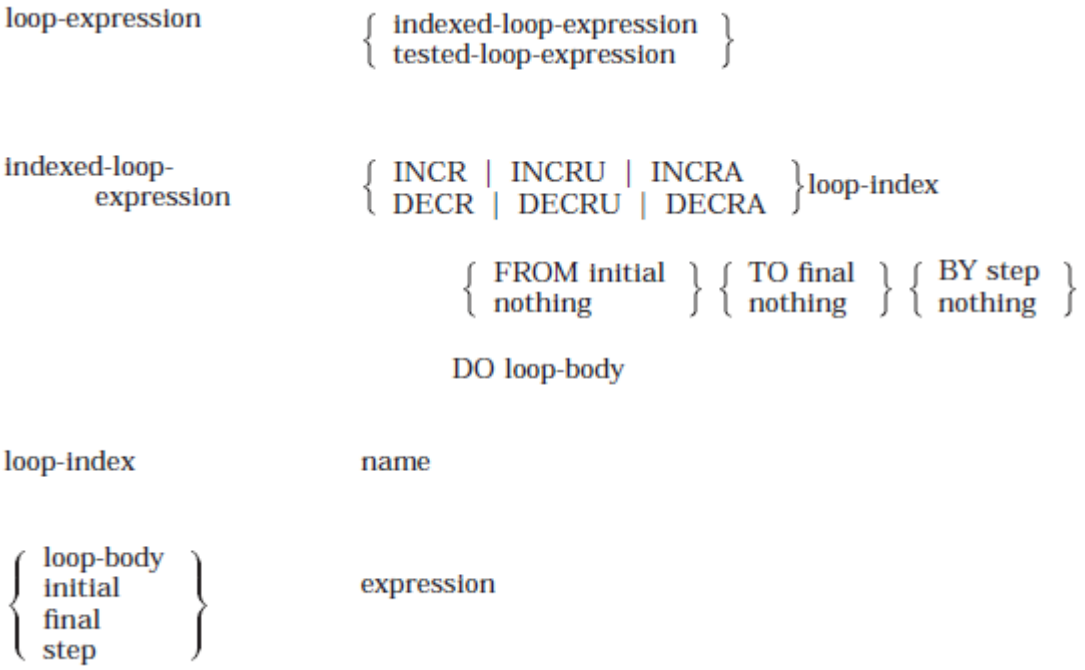
A loop-expression repeatedly evaluates a given expression, the loop-body. Loop-expressions are classified as indexed-loops (described in this section) and tested-loops (described in the next section).

An indexed-loop has a loop-index that starts at a given value and is stepped each time the loop cycles until a final value is reached. The loop-index not only determines the number of cycles performed by the loop, but can also be used as data in the calculations performed in the loop-body. An example of an indexed-loop follows:

```
OWN
    V: VECTOR[10],
    SUM;
    ...
SUM = 0;
INCR I FROM 0 TO 9 DO
    SUM = .SUM + .V[.I];
```

In this loop-expression, the loop-body is a single assignment-expression. The assignment-expression is evaluated ten times, for the sequence of values of .I as follows: 0, 1, 2, . . . , 9. The effect of the loop is to place the sum of the elements of the vector V in the data segment named SUM.

6.4.1. Syntax



6.4.2. Restrictions

The value of the step expression in an indexed-loop-expression must be positive.

6.4.3. Defaults

The initial, final, and step expressions can be omitted in an indexed-loop-expression. The following defaults apply:

Keyword	**Defaults**	
INCR	FROM 0 TO +infinity BY 1	
INCRU	FROM 0 TO +infinity BY 1	
INCRA	FROM 0 TO +infinity BY 1	
DECR	FROM largest-signed-value	TO 0 BY 1
DECRU	FROM largest-unsigned-value	TO 0 BY 1
DECRA	FROM largest-address-value	TO 0 BY 1

The default "+infinity" for INCR, INCRU, and INCRA loop-expressions means that no end test is made if no final expression is given. The "largest values" referred to are the maximum values accommodated by a signed or unsigned fullword, or the maximum address value provided, respectively, on the target system.

6.4.4. Semantics

The *loop-index* is implicitly declared to be a LOCAL name for the scope of the loop-body. This implicit declaration supersedes any previous declaration for that name throughout the indexed-loop. The MAP

declaration, described in *Section 10.10, "Map-Declarations"*, can be used to provide a structure attribute for the loop-index.

The keyword at the beginning of an indexed-loop-expression is INCR or DECR, followed by an optional added letter, U or A. The added letter affects the comparison of the index to the first and final expressions. Specifically:

No added letter:	Use signed comparison.
Last letter is U:	Use unsigned comparison.
Last letter is A:	Use address comparison.

Given the preceding discussion of indexes and keywords, the interpretation for an entire indexed-loop-expression is as follows:

1. Set the value of the loop-index to the value of the initial expression.
2. Evaluate the step and final expressions and save the values of these expressions.
3. If there is no final expression (so that "+infinity" is assumed by default), skip to step 5. Otherwise, perform the end test. The end test is satisfied if:
   - The keyword is INCR, INCRU, or INCRA, and the value of the loop-index is greater than the saved value of the final expression; or,
   - The keyword is DECR, DECRU, or DECRA and the value of the loop-index is less than the saved value of the final expression.
4. If the end test is satisfied, evaluation of the loop-expression is complete. Use –1 as the value of the loop-expression.
5. Evaluate the loop-body.
6. If the keyword is a form of INCR, add the saved value of the step expression to the loop-index. If the keyword is a form of DECR, subtract the saved value of the step expression from the loop-index. Go to step 3.

6.4.5. Pragmatics

The improper declaration of a loop-index is a common programming error. For example:

```
SUM = 0;
INCR I FROM 0 TO 9 DO
    BEGIN
    LOCAL
        I;
    SUM = .SUM + .V[.I];
    END;
```

The preceding program fragment is incorrect because I is used as a loop-index and then "blocked off" from use in the loop-body by an explicit declaration of I as LOCAL. The name I in .V[.I] refers to a data segment that is allocated by the explicit declaration, not to the implicit data segment that contains the loop-index. The correct version of this example appears at the beginning of this section (*Section 6.4, "Indexed-Loop-Expressions"*).

6.5. Tested-Loop-Expressions

A tested-loop-expression contains a test expression that is evaluated once during each loop cycle. The test expression determines whether or not repeated evaluation of the loop-body continues.

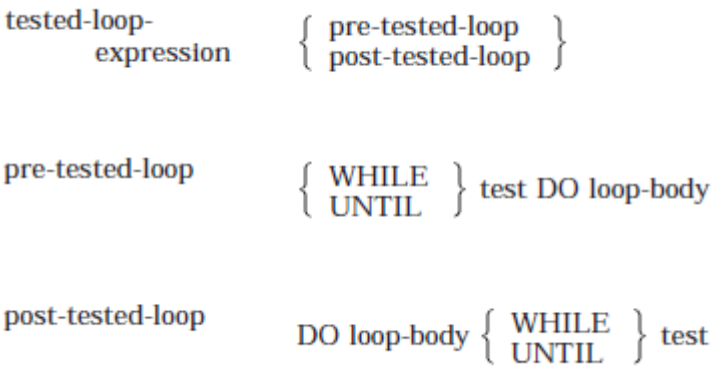
In a pre-tested loop, the test is made at the beginning of each cycle. If the test is satisfied, then the loop-body is evaluated and a new cycle begins; otherwise, evaluation of the loop-expression is complete. An example of a pre-tested-loop follows:

```
WHILE .PTR NEQ 0 DO
    BEGIN
    SUM = LIST[.PTR,CONT];
    PTR = LIST[.PTR,LINK];
    END;
```

In this example, the loop-body is the BEGIN-END block, with its two assignment-expressions. Each cycle of the loop begins with a test of the contents of PTR. If the value is not 0, then the block is evaluated and a new cycle begins; otherwise, evaluation of the loop-expression is complete.

A post-tested-loop differs from a pre-tested-loop only in the position of the test. In a post-tested-loop, the test is evaluated at the end of each cycle.

6.5.1. Syntax



6.5.2. Restrictions

The test in a pre-tested-loop or post-tested-loop is subject to the same evaluation rules as the test in a conditional-expression, described in *Section 6.1.3, "Semantics"*. Assignments or routine-calls that must be evaluated because they set values or have other side effects must not be included as part of a test.

6.5.3. Semantics

The interpretation of a pre-tested-loop is as follows:

1. Evaluate the test.

2. Examine the test clause (that is, the "WHILE test" or "UNTIL test"). The test clause is satisfied if the keyword is WHILE and the low-order bit of the test is 1 or if the keyword is UNTIL and the low-order bit of the test is 0.

3. If the test clause is satisfied, evaluate the loop-body and return to step 1.

4. If the test clause is not satisfied, use the value –1 as the value of the loop-expression.

The interpretation of a post-tested loop is as follows:

1. Evaluate the loop-body.

2. Evaluate the test.

3. Examine the test clause. If the test clause is satisfied, as defined in step 2 of the interpretation of the pre-tested-loop, return to step 1.

4. If the test clause is not satisfied, use the value –1 as the value of the loop-expression.

6.5.4. Pragmatics

The keywords WHILE and UNTIL are used to determine the continuation of a loop. If WHILE is used, then the loop continues if the low bit of the test expression value is 1. If UNTIL is used, the loop continues if the low bit of the test expression is 0. Thus, WHILE (test) is equivalent to UNTIL NOT (test).

The most fundamental form of loop is one that begins with the following:

```
WHILE 1 DO
```

Such a loop could cycle indefinitely because the loop test is always satisfied. Evaluation of the loop can be ended by an exit-expression (see *Section 6.6, "Exit-Expressions"*) or a return-expression (see *Section 6.7, "Return-Expressions"*) that is executed within the loop-body.

6.6. Exit-Expressions

An exit-expression gives three items of information: a command to end the evaluation of a block, the label of the block to which the command applies, and optionally a value for the designated block. An example of an exit-expression follows:

```
LEAVE ALPHA WITH .X-1;
```

This expression must occur in a block that is labeled ALPHA. It causes evaluation of that block to end and provides the value of .X–1 as the value of that block. The labeling of blocks is described in *Section 8.1, "Blocks"*.

6.6.1. Syntax

exit-expression { leave-expression | exitloop-expression }

leave-expression LEAVE label { WITH exit-value | nothing }

exitloop-expression EXITLOOP { exit-value | nothing }

label name

exit-value expression

6.6.2. Restrictions

A leave-expression must be contained in a block labeled by the same label that appears in the leave-expression.

An exitloop-expression must be contained in a loop-expression. If an exit-expression applies to an expression whose value is used, then the exit-expression must contain an exit-value.

6.6.3. Semantics

The semantics of the two kinds of exit-expression is presented in the following sections.

6.6.3.1. Leave-Expressions

The interpretation of a leave-expression follows:

1. If an exit-value is given, evaluate the exit-value and use that value as the value of the labeled-block.
2. If an exit-value is not given, the value of the labeled-block is undefined.
3. End the evaluation of the labeled-block designated by the label of the leave-expression.

6.6.3.2. Exitloop-Expressions

The interpretation of an exitloop-expression follows:

1. If an exit-value is given, evaluate the exit-value and use that value as the value of the loop-expression.
2. If an exit-value is not given, the value of the loop-expression is undefined.
3. End the evaluation of the innermost loop.

6.6.4. Pragmatics

An exitloop-expression is a special case of a leave-expression that leaves the innermost containing loop-expression. An exitloop-expression is convenient because it does not require the use of a label.

An example of an exitloop-expression appears in the following program fragment:

```
OWN
    X: VECTOR[10],
    ZEROFLAG;
...
ZEROFLAG = 0;
INCR I FROM 0 TO 9 DO
    IF .X[.I] EQL 0
    THEN (ZEROFLAG = 1; EXITLOOP);
```

The elements of the vector X are examined to determine if there is an element whose contents is 0. If an element containing 0 is found, then ZEROFLAG is set to 1 and evaluation of the loop-expression is ended by the EXITLOOP. Evaluation of the loop ends when the first zero is found; the elements of the vector following the first element containing 0 are not examined. An example of a leave-expression appears in the following program fragment:

```
OWN
    XYZ: ARRAY[10,20],
    ZEROFLAG;
LABEL
    L;
...
ZEROFLAG = 0;        ! Initialize to no zeros found
L:  BEGIN
    INCR I FROM 0 TO 9 DO
    INCR J FROM 0 TO 19 DO
        IF .XYZ[.I,.J] EQL 0
            THEN (ZEROFLAG = 1; LEAVE L);
    END;
```

When the leave-expression is evaluated, it ends evaluation of two loops: the inner loop with index J and the outer loop with index I.

The value of an exit-expression can be used to give a value to a loop. An example of this use of an exit-expression appears in the following program fragment:

```
OWN
    VALBUF: VECTOR[10],
    BUFLEN;
...
BUFLEN = 1+
    BEGIN
    DECR J FROM 9 TO 0 DO
        IF .VALBUF[.J] NEQ 0 THEN EXITLOOP .J
    END;
```

Assume that the initial elements of VALBUF contain nonzero values, and the remaining elements contain zero. BUFLEN is the number of nonzero values in VALBUF. Observe that if a nonzero value is found, then the exitloop-expression ends the evaluation of the loop. If the buffer is all zeros, the evaluation of the loop runs to completion and the loop value is –1. In both cases, the value returned is 1 less than the desired number of values.

6.7. Return-Expressions

A return-expression is used to end the evaluation of a routine and send control back to the point at which the routine was called.

6.7.1. Syntax

return-expression — RETURN { returned-value | nothing }

returned-value — expression

6.7.2. Restrictions

A return-expression in a routine that does not have the NOVALUE attribute must have a returned-value.

6.7.3. Semantics

The interpretation of the return-expression follows:

1. If the return-expression has a returned-value, evaluate the returned-value and use that value as the value of the routine-body.
2. End the evaluation of the routine-body.

Discussion of return-expressions is presented in the sections on the NOVALUE attribute (*Section 9.10, "The Novalue-Attribute"*) and routine-declarations (*Section 12.3, "Routine-Declarations"*).

Chapter 7. Constant Expressions

A *constant expression* is an expression that can be evaluated before program execution begins. The practical and efficient implementation of BLISS requires that constant expressions be used in certain contexts, as specified in the syntax diagrams. An expression is a constant expression if certain restrictions are met, and those restrictions are given in this chapter.

There are two kinds of constant expression. The compile-time constant expression is the more heavily restricted of the two, and can be evaluated during the compilation of the module in which it appears. The link-time constant expression includes the compile-time constant expression as a special case, and can be evaluated by the compiler, the linker, and the operating system working together.

This chapter has two sections, one for each kind of constant expression.

7.1. Compile-Time Constant Expressions

This section defines *compile-time constant expressions*. The definition assumes the definition of expressions given in the previous chapters and then imposes restrictions. The restrictions are designed to permit a compile-time constant expression to be evaluated during the compilation of the module in which it appears. When the compiler encounters a compile-time constant expression, it evaluates that expression and makes use of its value in compiling efficient object code.

Constant values known to the compiler are required in several places in BLISS in order to give a reasonable interpretation to another language feature. For example, in order for the compiler to allocate static storage for plits, the actual sizes of all components must be known – including any repetition counts. The same consideration applies to the sizes of other static storage declarations, such as an own-declaration.

In other cases, requiring constant values assures that an efficient implementation can be provided by the compiler. For example, requiring that all LOCAL (and STACKLOCAL) storage allocation is of constant size and therefore known to the compiler assures that storage allocation can be done efficiently and that LOCAL data segments can be addressed efficiently.

Some simple examples of compile-time constant expressions are as follows:

```
5
3 * 15 - 4
7 + %C'A'
MAX(3, 7, 3*15-4)
```

Compile-time constant expressions often involve names that are declared LITERAL; for example:

```
LITERAL
    REG = 5,
    SIZE = 47;
    ...
    BEGIN
    OWN X: VECTOR[MAX(SIZE,3)+1];
    REGISTER A = REG;
    ...
    END
```

There are quite a few contexts that require compile-time constant expressions, and they are scattered through the language. For convenience, a complete list follows.

A compile-time constant expression must be used as follows:

- The *replicator* in a PLIT (*Chapter 4, "Primary Expressions"*)
- The *low-bound*, *high-bound*, *single-value*, *low-value*, and *high-value*

  expressions in a case-expression (*Chapter 6, "Control Expressions"*)
- The *boundary* expression in an alignment-attribute (*Chapter 9, "Attributes"*)
- The *ctce-access-actual* in a preset-attribute of a data-declaration (*Chapter 9, "Attributes"*)
- The *bit-count* in a range-attribute of a literal- or external-literal-declaration (*Chapter 9, "Attributes"*)
- The *register-number* in a register-declaration (*Chapter 10, "Data Declarations"*)
- The *sign-extension-flag* in a field-selector (*Chapter 11, "Data Structures"*)
- The *structure-size* in the declaration of a structure-name (*Chapter 11, "Data Structures"*)
- The *allocation-actual* parameter in a structure-attribute (*Chapter 11, "Data Structures"*)
- The *field-component* in a field-declaration (*Chapter 11, "Data Structures"*)
- The *register-number* in a linkage-option (*Chapter 13, "Linkages"*)
- The *literal-value* in a literal-declaration (*Chapter 14, "Binding"*)
- Certain parameters in lexical-functions (*Chapter 15, "Lexical Functions"*)
- The *lexical-test* in a lexical-conditional (*Chapter 15, "Lexical Functions"*)
- The *compile-time-value* in a compile-time-declaration (*Chapter 15, "Lexical Functions"*)
- The level value in an OPTLEVEL module-switch (*Chapter 19, "Modules and Programs"*)

7.1.1. Syntax

compile-time-constant-expression ::= expression

7.1.2. Restrictions

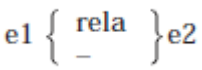
These restrictions apply to an expression after any macro calls in the expression have been expanded.

A compile-time constant expression must be one of the following expressions:

- A *numeric-literal*.
- A *string-literal*.
- A *name* that satisfies the following conditions:
  - It is declared in any bound-declaration except an EXTERNAL literal-declaration (as described in *Chapter 14, "Binding"*).

    - It is bound to a value that is given by a compile-time constant expression.

- A *structure-reference* that yields a compile-time constant expression when it is expanded (as described in *Chapter 11, "Data Structures"*).
- A *block* that has a compile-time constant expression (and nothing else) as its body.
- An *operator-expression* that satisfies the following conditions:
    - It is not a fetch-expression or an assignment-expression.
    - It has a compile-time constant expression as each of its operands.
- An *operator-expression* that has the following form:



  In these forms, *rela* is one of the relational operators for addresses (EQLA, NEQA, and so on). Both *e1* and *e2* must be link-time constant expressions; furthermore, their values must be addresses that are relative to the same program section, external data segment, or external routine name.

- An *executable-function* that satisfies the following conditions:
    - It is the ABS function, the SIGN function, or one of the max or min functions.
    - It has a compile-time constant expression as each of its parameters.
- A *supplementary-function* that satisfies certain restrictions. Those restrictions are not given here but instead appear as part of the definition of each supplementary-function. For example, *Section 20.2.1.1, "Definition"* states that the CH$ALLOCATION function is a compile-time constant expression if its parameters are compile-time constant expressions.
- A *conditional-expression* that satisfies the following conditions:
    - It has a test that is a compile-time constant expression.
    - It has a consequence or alternative that is a compile-time constant expression, depending on whether the test is satisfied or fails.
- A *case-expression* that satisfies the following conditions:
    - It has a *case-index* that is a compile-time constant expression.
    - It has at least one case-action that is a compile-time constant expression; namely, that case-action that is chosen by the value of the case-index.

7.1.3. Semantics

A compile-time constant expression is evaluated during the compilation of the module in which it appears. In all other respects, its interpretation is the same as that for an unrestricted expression (see *Chapter 4, "Primary Expressions"*, *Chapter 5, "Computational Expressions"*, and *Chapter 6, "Control Expressions"*).

7.2. Link-Time Constant Expressions

This section defines *link-time constant expressions*. The definition assumes the definition of expressions given in the previous chapters, and then imposes restrictions. The definition of link-time constant expressions includes the compile-time constant expressions as a special case. The restrictions on a link-time constant expression are designed to permit the expression to be evaluated by the compiler, the linker, and the operating system before the value is needed for program execution.

The need for link-time constant expressions arises in two ways:

- A name that designates storage in a program section is specified as an offset, not a full, absolute address, by the compiler. The absolute address cannot be determined until link time, when the program sections are allocated and their base addresses are determined.
- A name that is declared EXTERNAL is entirely undetermined at compile time because its original declaration is in another module. Its offset or its absolute address cannot be determined until link time, when the module in which the GLOBAL declaration of the name appears is present.

A simple example of the use of a link-time constant expression is contained in the following program fragment:

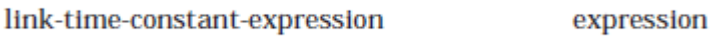
```
OWN X: VECTOR[10];
...
OWN ALPHA: INITIAL(X[2]);
```

During compilation, the final value of X is not known; it is expressed as an offset in the OWN program section. Only at link time is it possible to determine the absolute address of X, to evaluate X[2] (the address of the third element of X), and, finally, to supply the initial value for ALPHA.

There are five contexts in which a link-time constant expression is required:

- The *plit-expression* in a PLIT (*Chapter 4, "Primary Expressions"*)
- The *plit-expression* in an initial-attribute of an own- or global-declaration (*Chapter 9, "Attributes"*)
- The *preset-value* in a preset-attribute of an own- or global-declaration (*Chapter 9, "Attributes"*)
- The *data-name-value* in a GLOBAL bind-data-declaration (*Chapter 14, "Binding"*)
- The *routine-name-value* in a GLOBAL bind-routine-declaration (*Chapter 14, "Binding"*)

7.2.1. Syntax



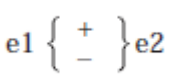
7.2.2. Restrictions

These restrictions apply to an expression after any macro-calls in the expression have been expanded.

A link-time constant expression must be one of the following expressions:

- A *compile-time-constant-expression*.
- A *PLIT*.

- A *name* that is declared as one of the following:
  - OWN, GLOBAL, EXTERNAL, or FORWARD. These are used for names of permanently allocated data segments.
  - ROUTINE, GLOBAL ROUTINE, EXTERNAL ROUTINE, or FORWARD
  - ROUTINE. These are used for names of routine segments.
  - EXTERNAL LITERAL. This is used for names of literals that are bound in other modules.
- A *name* that satisfies the following conditions:
  - It is declared by a bound-declaration (as described in *Chapter 14, "Binding"*).
  - It is bound to a value that is given by a link-time-constant-expression.
- A *structure-reference* that yields a link-time constant expression when it is expanded (as described in *Chapter 11, "Data Structures"*).
- A *block* that has a link-time constant expression (and nothing else) as its body.
- An *operator-expression* that has the following form:

  In these forms, *e1* must be a link-time constant expression and *e2* must be a compile-time constant expression.
- An *operator-expression* that has the following form:

  In these forms, *rela* is one of the relational operators for addresses (EQLA, NEQA, and so on). Both *e1* and *e2* must be link-time constant expressions; furthermore, their values must be addresses that are relative to the same program section, external data segment, or external routine name.
- A *supplementary-function* that satisfies certain restrictions. Those restrictions are not given here but appear as part of the definition of each supplementary function. For example, *Section 20.2.2.1, "Definition"* states that the CH$PTR function is a link-time constant expression if its first parameter is a link-time constant expression and its remaining parameters are compile-time constant expressions.

7.2.3. Semantics

A link-time constant expression is evaluated during the compilation, linking, and loading of the module in which it appears. In all other respects, its interpretation is the same as that for an unrestricted expression (see *Chapter 4, "Primary Expressions"*, *Chapter 5, "Computational Expressions"*, and *Chapter 6, "Control Expressions"*).

The following summarizes the restrictions presented above:

A link-time-constant-expression is one of the following:

- Any compile-time constant expression
- A data segment name or external name
- A data segment name or external name modified by adding or subtracting a constant value (using + and –)
- The result of comparing or taking the difference of two link-time constant expressions that represent addresses in the same program section or relative to the same external name (using the relational operators for addresses)

Chapter 8. Blocks and Declarations

This chapter describes the general structure and use of blocks and declarations. Blocks and declarations are the fundamental structural features of BLISS. They are interdependent and complementary. A block is used to gather a sequence of declarations and expressions into a single construct. In contrast, a declaration is used to distribute a single set of information to many places in a block, that is, to each place where the declared name is used.

Later chapters describe specific types of declarations in detail.

8.1. Blocks

On the inside, a block can contain a long and complicated sequence of declarations and expressions. From the outside, that same block is a single syntactic unit that has a single value. In this way, blocks provide for the large-scale structuring of a program.

Blocks need not be complicated. They are often used to specify the order in which operators are to be evaluated; for example:

```
2*(.A-1)
```

In this expression, (.A-1) is a block. It is used to show that the difference of .A and 1 should be calculated before multiplication by 2. This block is the simplest kind of block, a *parenthesized expression*.

In some cases, a block is used to gather several expressions together so that they are evaluated as a unit; for example:

```
IF .ALPHA NEQ 0
THEN
    BEGIN
    Q1 = .ALPHA*.S1;
    Q2 = .ALPHA*.S2;
    END;
```

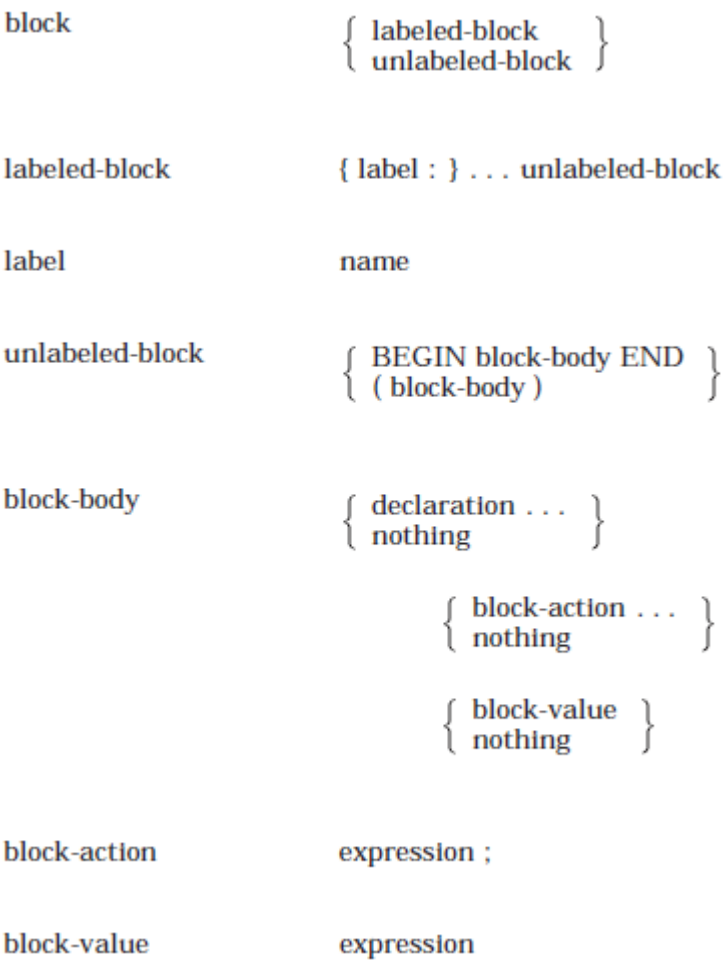
An equivalent way of writing this block follows:

```
IF .ALPHA NEQ 0 THEN (Q1 = .ALPHA*.S1; Q2 = .ALPHA*.S2;);
```

The block in these examples is a *compound-expression*; that is, a block that contains one or more expressions but does not contain a declaration. The choice between parentheses and the BEGIN-END pair is entirely a matter of appearance and readability.

Finally, a block can be used to gather together a sequence of declarations and expressions of arbitrary length and complexity.

8.1.1. Syntax



A block *immediately contains* a given construct (such as a name or a declaration) if it is the *smallest* block that contains the given construct.

A *compound-expression* is a block that does not immediately contain any declarations.

A *parenthesized-expression* is a block that has the following form:

```
(expression)
```

8.1.2. Restrictions

The label in a labeled-block must be declared by a label-declaration (see *Section 18.4, "Label-Declarations"*).

A block that appears in a context that requires a value must contain a block-value expression.

A block must not be empty; that is, it must contain at least one declaration, block-action, or block-value.

8.1.3. Semantics

Consider, first, a block whose evaluation runs to completion without being prematurely ended by, for example, a leave-expression. The block is evaluated in three steps, as follows:

1. Process the declarations (if any).

2. Evaluate the block-actions (if any) in the order in which they are written.

3. Evaluate the block-value expression (if any).

If the block has a block-value expression, then the value of that expression is the value of the block; otherwise, the value of the block is undefined and an attempt to use that value is invalid.

Most of the processing of declarations is performed before program execution begins. For example, the information in an OWN declaration is used by the compiler and linker to allocate storage, provide an initial value, and so on. In a few cases, the processing of a declaration requires run-time calculations. For example, the value in a BIND declaration can be given by an expression that must be evaluated each time the block is entered.

The evaluation of block-actions in order, one after another, is the basis for sequential flow of control. It is valid to assume that the evaluation of a block-action is completed before the evaluation of the next block-action begins. In the course of optimization, the compiler alters the order of some calculations, but never in a way that affects the results.

In BLISS the block-action plays a role similar to the role of the "statement" in other high-level languages. The semicolon at the end of a block-action has the syntactic role of separating the block-action from the next component of the block. In addition, it has the semantic effect of discarding the value of the expression. Thus it is valid to use an expression whose value is undefined as the expression in a block-action.

Consider, next, a block that does not run to completion. Such a situation arises because of a return-expression, leave-expression, or exitloop-expression that is contained in the block. In this situation, the value of the block is the value supplied by the return-expression, leave-expression, or exitloop-expression. If no value is supplied, then the value of the block is undefined.

8.1.4. Discussion

An example of a block is contained in the following conditional-expression:

```
IF .Q EQL 0
THEN
    BEGIN
    LOCAL
        TEMP;
    TEMP = .X;
    X = .Y;
    Y = .TEMP;
    END;
```

The block is evaluated if the contents of Q is 0.

The block in this example begins with one declaration, continues with three block-actions, and does not contain a block-value expression. The declaration describes a data segment named TEMP, which is allocated for use in this block only. The block actions are all assignments; they exchange the contents of X and Y. Clearly, it is important, in this example, that the assignments are performed in the order written.

The entire example is an expression (a conditional-expression) followed by a semicolon. Therefore, it is a block-action and is part of some larger block (not shown).

8.2. Declarations

A declaration provides information about the block that contains it. Usually, the information affects the interpretation of one or more names that are used in the block. Thus, although the declaration does not directly cause any action, it does affect the interpretation of the block by specifying information about the names that are declared.

In the simplest case, the information provided by a declaration is just a single keyword; for example:

```
OWN
    X;
```

This specifies that X is an OWN name.

Sometimes a declaration gives some of the attributes that are described in *Chapter 9, "Attributes"*. For example:

```
GLOBAL
    DELTA: VECTOR[120] INITIAL(REP 120 OF (-1));
```

This specifies that DELTA is a GLOBAL name and that it has the given structure- and initial-attributes.

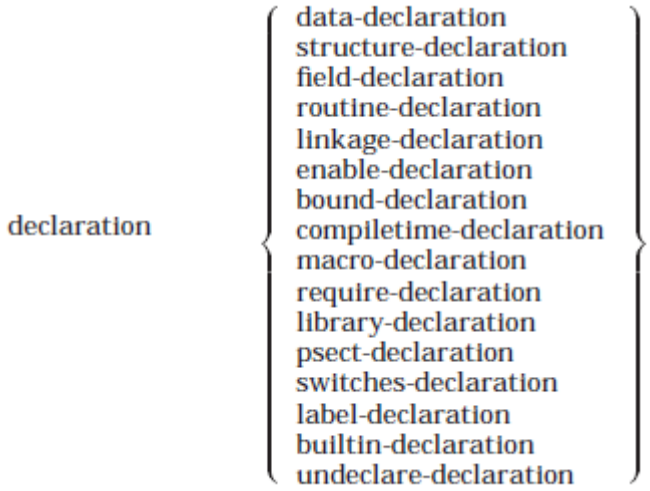
In other cases, a declaration can give even more information. For example:

```
GLOBAL ROUTINE EXCH(X,Y): NOVALUE =
    BEGIN
    LOCAL TEMP;
    TEMP = ..X;
    .X = ..Y;
    .Y = .TEMP;
    END;
```

This specifies that EXCH is a global routine-name, that it has the novalue-attribute, that it has the formal-name list (X,Y), and that it designates the routine given in the BEGIN-END block.

A declaration applies to those occurrences of a name that are within its scope. In the example just given, the declaration (LOCAL TEMP;) applies only to the occurrences of TEMP within the BEGIN-END block. The example is part of a module (not shown), but any other use of TEMP in that module lies outside the scope of the local-declaration in the example.

8.2.1. Syntax



The syntax diagrams for the specific kinds of declarations are given in later chapters. With few exceptions, however, each kind of declaration declares a user-chosen symbol as a specific kind of name (data-segment name, structure-definition name, routine name, and so forth), and generally provides additional information about that name.

A given name can be used more than once in a module and can have different declarations in different places. The declaration that applies to a given use of a name *governs* that name. To find the declaration that governs a given use of a name, proceed as follows:

Start at the given use of the name and scan backwards through the module. If the end of a block is encountered, skip over everything contained in that block. The first declaration of the given name that is encountered during this scan is the desired declaration.

One declaration of a name can govern many uses of the name. The part of a module that is governed by a declaration is the *scope* of that declaration.

8.2.2. Restrictions

Every use of a name must be governed by an explicit declaration. The predeclared names (see *Appendix A, "Predefined Identifiers"*) are an exception to this rule; they can be used without being explicitly declared.

Two declarations of the same name must not be immediately contained in the same block.

The two restrictions just given are subject to some exceptions when UNDECLARE declarations are used (see *Chapter 18, "Special Features"*).

A name is declared as *global* when its declaration begins with the keyword GLOBAL. A name must not be declared global more than once in a program.

8.2.3. Semantics

A declaration supplies the following information about each occurrence of a name that it governs:

1. The one or more keywords with which the declaration begins

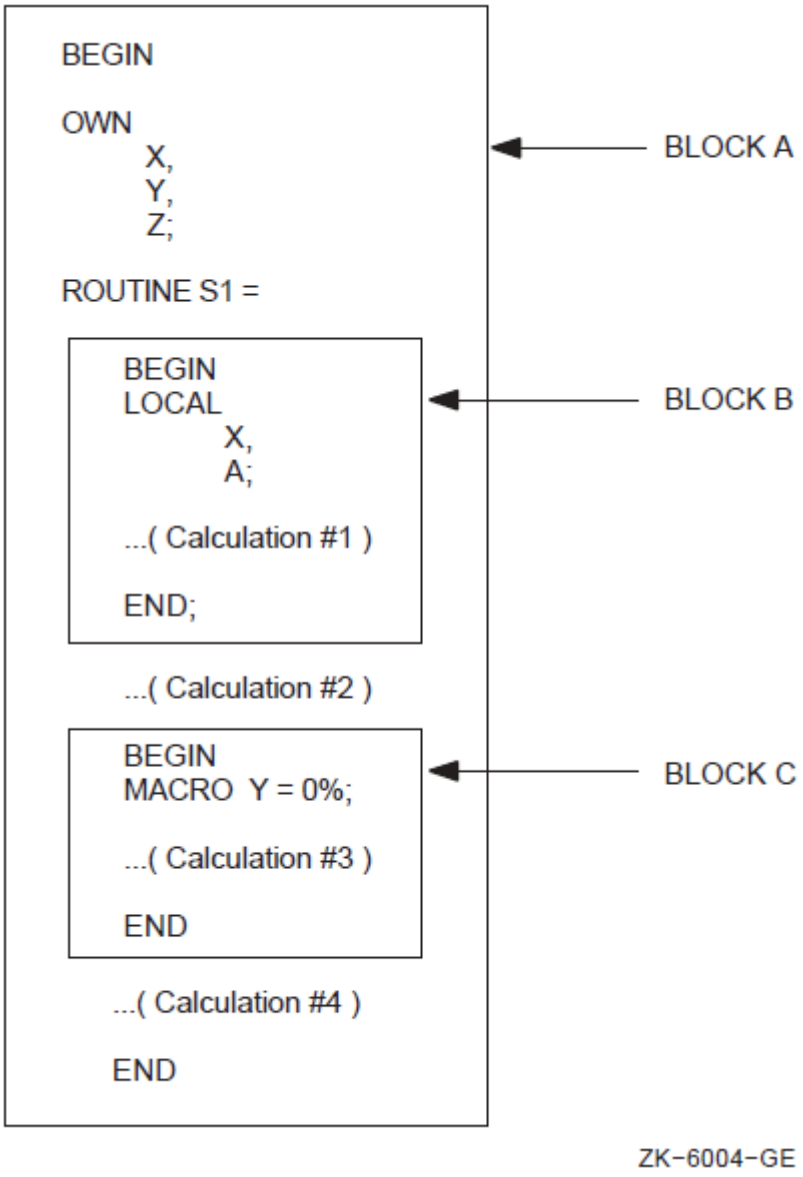
2. The attributes that appear in the declaration of the name

3. Other, specialized, information that is included in certain kinds of declaration, such as the routine-body in a routine-declaration, or the bound-value in a bind-declaration

Most of the information supplied by the declaration is processed by the compiler. For most declarations, part of the processing defines a value for the declared name. For example, when an own-declaration is processed, an address offset is associated with the name, and that address-offset is bound (by the linker) to the address of a data segment.

8.2.4. Discussion

As defined in *Section 8.2.1, "Syntax"*, the scope of a declaration is the part of a module that is governed by the declaration. An example of scopes is given in the following diagram:



The three blocks in this example are enclosed in boxes that are identified as A, B, and C for convenience of discussion. Block A designates the entire example (including the contents of Block B and Block C). The details of the calculations performed by the example block are not important, so they are omitted.

The places where names could be used in calculations are called Calculation #1, Calculation #2, and so on.

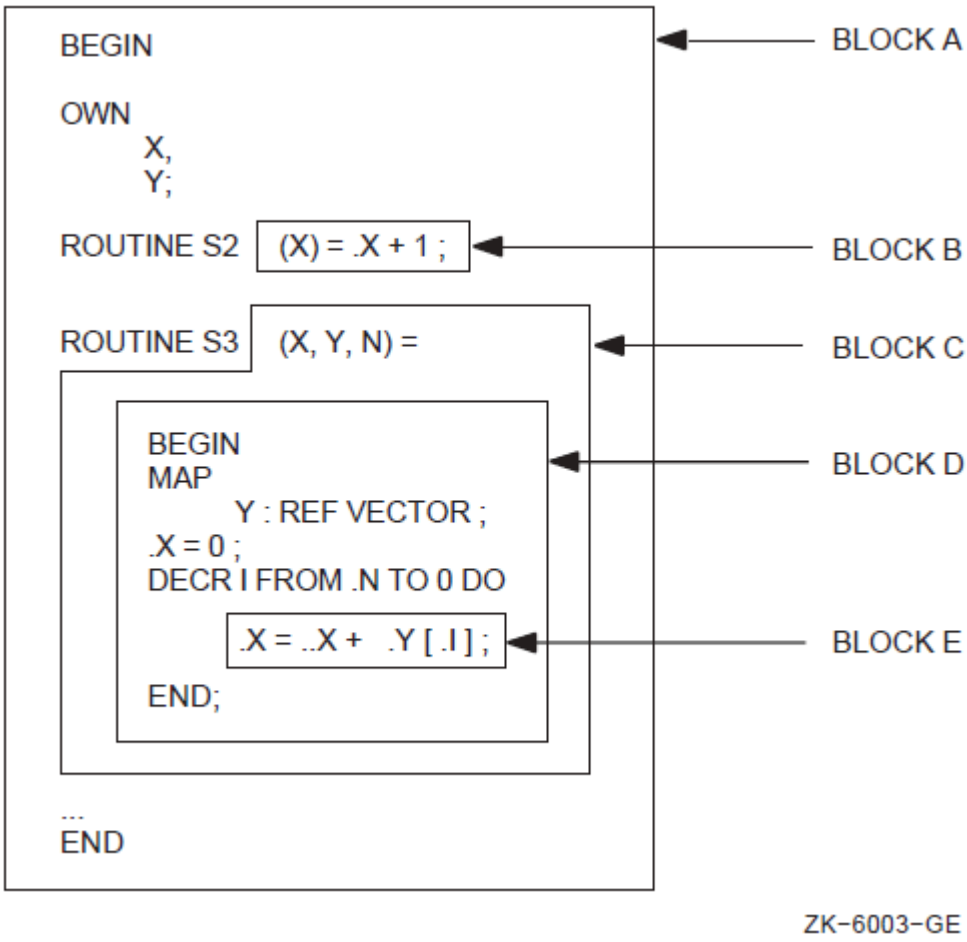
The example contains seven declarations of names. The scopes of the declarations are:

Declaration		**Scope of Declaration**
X	(in Block A)	Block A except Block B
Y	(in Block A)	Block A except Block C
Z	(in Block A)	Block A
S1	(in Block A)	Block A
X	(in Block B)	Block B
A	(in Block B)	Block B
Y	(in Block C)	Block C

Another way to express this information is to show the declaration that governs each name in each of the calculations, as follows:

Use of Name	**Declaration of Name**	
In Calculation #1	LOCAL	(Block B)
X	OWN	(Block A)
Y	OWN	(Block A)
Z	ROUTINE	(Block A)
S1	LOCAL	(Block B)
A		
In Calculation #2	OWN	(Block A)
X	OWN	(Block A)
Y	OWN	(Block A)
Z	ROUTINE	(Block A)
S1	(undeclared)	
A		
In Calculation #3	OWN	(Block A)
X	MACRO	(Block C)
Y	OWN	(Block A)
Z	ROUTINE	(Block A)
S1	(undeclared)	
A		
In Calculation #4	(Same as in Calculation #2)	

A second example of scope follows:



The blocks in this example are labeled in the same way as in the previous example. Three of the blocks are implicit; that is, they are assumed to exist even though a BEGIN-END or parenthesis pair is not used. Specifically, Blocks B and C are the implicit blocks that each surround the formal-names and the routine-body of a routine-declaration. Block E is the implicit block that surrounds the body of a loop.

This example contains ten declarations. Five of the declarations are implicit. Specifically, the formal-name X is implicitly declared in Block B; the formal-names X, Y, and N are implicitly declared in Block C; and the loop-index I is implicitly declared in Block E. The scopes of the declarations are as follows:

Declaration		**Scope of Declaration**
X	(in Block A)	Block A except Blocks B and C
Y	(in Block A)	Block A except Block C
S2	(in Block A)	Block A
X	(in Block A)	Block B
S3	(in Block B)	Block A
X	(in Block A)	Block C
Y	(in Block C)	Block C except Block D
N	(in Block C)	Block C
Y	(in Block C)	Block D

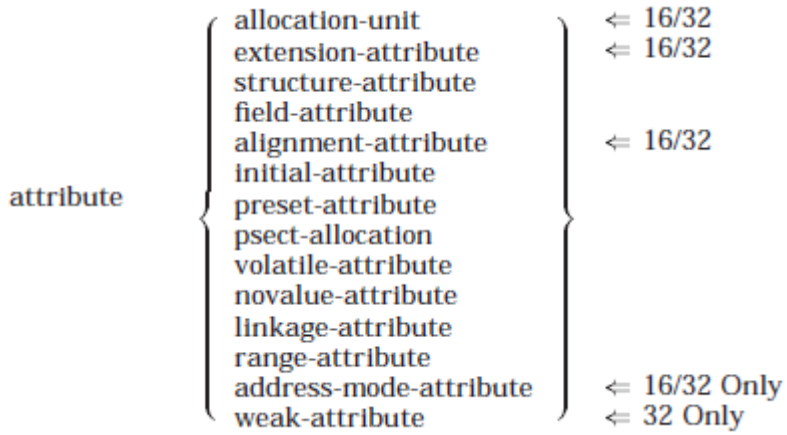
I	(in Block D)	Block E
	(in Block E)	

Unlike all other declarations, the MAP declaration *redeclares* a name; that is, it establishes a new set of attributes to be used with a previously declared data segment name. Thus, the two declarations of Y in Blocks C and D refer to the same data segment.

Chapter 9. Attributes

Many declarations are used to associate attributes with a declared name, as well as declaring the name to be of a specific kind. Some attributes are common to many forms of declarations, and some apply to only a few forms. This chapter describes the attributes themselves.

The following syntax diagram lists the attributes:



Each attribute is described in a section of this chapter. A final section summarizes the usage of attributes by showing which attribute can be used with which kind of declaration.

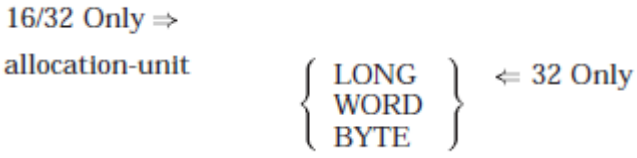
9.1. The Allocation-Unit – BLISS–16/32 Only

An allocation-unit can be used in a data-declaration or a bind-data-declaration. An allocation-unit can appear either as an independent attribute or as an allocation-actual parameter within a structure-attribute (as described in *Chapter 11, "Data Structures"*).

An allocation-unit is used wherever the "granularity" of storage allocation must be specified. Examples of the use of allocation-units in the declaration of names follow:

```
OWN                         !A is a scalar data segment composed
    A: WORD;                !of one word (16 bits).

GLOBAL                      !B is a vector data segment composed
    B: VECTOR[10,BYTE];     !of ten one-byte elements.

LOCAL                       !C is a scalar data segment composed
    C;                      !(by default) of one fullword.
```

9.1.1. Syntax



9.1.2. Default

The default allocation-unit is WORD for BLISS–16, and LONG for BLISS–32.

9.1.3. Restriction

As shown in the syntax diagram, the allocation-unit LONG is valid for BLISS–32 only.

An allocation-unit (used as an attribute) must not be used in the same declaration as a structure-attribute.

If a declaration contains both an allocation-unit (used as an attribute) and an initial-attribute, then the allocation-unit must precede the initial-attribute.

9.1.4. Semantics

An allocation-unit specifies a quantity of storage, as follows:

LONG	32 bits
WORD	16 bits
BYTE	8 bits

If the declaration of a name does not contain a structure-attribute (and is therefore a scalar declaration), the allocation-unit determines the quantity of storage allocated for the entire data segment. If the declaration has a structure-attribute, the attribute can include an allocation-unit as one of its allocation-actuals.

9.2. The Extension-Attribute – BLISS–16/32 Only

Like an allocation-unit, an extension-unit can be used in a data-declaration or a bind-data-declaration. An extension-attribute can appear either as an independent attribute or as an allocation-actual within a structure-attribute (as described in *Chapter 11, "Data Structures"*).

Examples of the use of an extension-attribute follow:

```
OWN                                 !A is a scalar data segment composed
    A: SIGNED WORD;                 !of one signed word.

GLOBAL                              !B is a vector data segment composed
    B: VECTOR[10,BYTE,SIGNED];      !of 10 signed bytes.

LOCAL                               !C is a scalar segment composed
    C: UNSIGNED BYTE;               !of one unsigned byte.
```

9.2.1. Syntax

9.2.2. Restriction

An extension-attribute (used as an attribute) must not appear in the same declaration as a structure-attribute.

9.2.3. Default

The default extension-attribute is UNSIGNED.

9.2.4. Semantics

An extension-attribute specifies the value extension rule to use when fetching the contents of a scalar field value. SIGNED specifies that the high order bit of the fetched value (the sign bit) is to be used. UNSIGNED specifies that zero bits are to be used.

The extension-attribute is normally specified in combination with the allocation-unit BYTE in BLISS–16, and with BYTE or WORD in BLISS–32.

9.3. The Structure-Attribute

A structure-attribute can be used in a data-declaration or a bind-data-declaration. It associates the declared data-segment name to a separately declared structure-definition, causing the allocation of the data-segment to be controlled by that structure-definition. Subsequent access to the data-segment is also controlled by the associated structure-definition. A structure-definition is declared in a structure-declaration. BLISS provides several predeclared structure-definitions, as described in *Chapter 11, "Data Structures"*.

An example of the use of a structure-attribute follows:

```
OWN
    X: VECTOR[8];
```

The structure-attribute here is VECTOR[8]. The attribute specifies that X is a data-segment with a VECTOR structure. The predeclared structure-definition named VECTOR is described in *Section 11.10, "Predeclared Structures"*. In accordance with that definition plus the allocation-actual, 8, specified in the attribute, X is allocated as a sequence of eight fullword elements that are designated X[0] through X[7]. In BLISS–16 or BLISS–32, an allocation-unit can be used as an additional allocation-actual. For example, VECTOR[8,BYTE], to specify the size of the elements allocated.

A structure-attribute can name a user-declared structure-definition as well as one of the standard, predeclared structures described in *Chapter 11, "Data Structures"*. In any case, the interpretation of the structure-attribute depends entirely on the structure-declaration that governs the given structure-name.

For example:

```
GLOBAL
    Y: MATRIX[10];
```

The structure-attribute here is MATRIX[10]. The attribute specifies that Y is a MATRIX structure. BLISS does not have a predeclaration for the name MATRIX; therefore, this example must occur in the scope of an explicit STRUCTURE declaration of MATRIX. The interpretation of the example depends entirely on that STRUCTURE declaration.

The structure-attribute is fully described in *Chapter 11, "Data Structures"*, together with the structure-declaration.

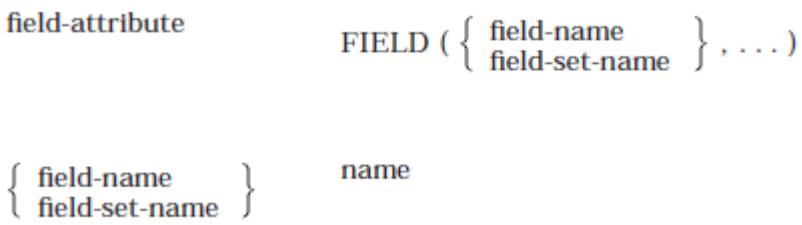
9.4. The Field-Attribute

A field-attribute can be used in data-declarations and bind-data-declarations. It specifies one or more *field-names* that are to be associated with the declared data-segment-name. This association allows the field-names to be used in structure-references to the data segment, as described in *Chapter 11, "Data Structures"*. The field-attribute is meaningful only in declarations of structured data segments.

The definition of a field-name, in terms of field-component values, is given in a field-declaration that governs the use of that name. Field-declarations are also described in *Chapter 11, "Data Structures"*.

For notational convenience, a group of field-name definitions can be identified (in the field-declaration) by a field-set-name and can then be referred to in a field-attribute by that single name.

9.4.1. Syntax



9.4.2. Default

If a field-attribute is not specified for a data-segment-name, no field-names may appear in an ordinary-structure-reference to the corresponding data segment.

9.4.3. Semantics

A field-attribute specifies the set of field-names that can validly appear in an ordinary-structure-reference to a data segment declared with the given field-attribute. A field-set-name in a field-attribute specifies a set of field-names that can so appear. If no field-attribute is given, then no field-name is valid in such a reference.

9.5. The Alignment-Attribute – BLISS–16/32 Only

An alignment-attribute can be used in an OWN, GLOBAL, LOCAL, or STACKLOCAL data-declaration. In BLISS–32, an alignment-attribute can also be used in a psect-declaration, as described in *Section 18.1.1, "Syntax"*. This attribute indicates the address alignment required for a data segment relative to the different levels of address boundaries (for example, byte, word, longword, quadword).

The purpose of the alignment-attribute is to specify the smallest boundary at which the data segment may be allocated, generally a larger boundary than the default one. For example, an alignment-attribute

might be used to specify that a particular byte-scalar segment is to start at a word boundary only, rather than at any byte boundary which is the default. Use of this attribute can result in unused storage left between the previously allocated data segment and the data segment to which the attribute applies.

The alignment-attribute indicates a particular address boundary by means of a boundary value, *n*, which specifies that the binary address of the data segment must end in at least *n* 0s. For example:

```
OWN
    A:BYTE ALIGN(1);
```

The alignment-attribute, ALIGN(1), specifies that data-segment A is to be allocated at an address that ends with at least one 0; which is to say that it is to be aligned to a word boundary.

An example of BLISS–32 usage of the alignment-attribute is given in *Section 9.5.5, "Discussion"*.

9.5.1. Syntax

16/32 Only ⇒

alignment-attribute	ALIGN (boundary)
boundary	compile-time-constant-expression

9.5.2. Restrictions

The value of boundary must be a positive integer.

BLISS–16 ONLY

The value of boundary must be either 0 or 1, corresponding to byte- or word-boundary alignment respectively.

The value of boundary must not exceed the value of the program-section alignment boundary for the storage class being allocated.

The value of boundary in a LOCAL or STACKLOCAL declaration must not exceed 2.

9.5.3. Default

The default alignment depends on the kind of data that is declared, as follows:

Kind of Data	**Default Alignment**
BYTE scalar	ALIGN(0)
WORD scalar	ALIGN(1)
LONG scalar	ALIGN(2) – 32 Only
Any structure	ALIGN(1) – 16 Only
Any structure	ALIGN(2) – 32 Only

9.5.4. Semantics

Suppose the value of the boundary expression is *n*. The compiler allocates the declared data segment in the unused portion of the appropriate program section at the smallest possible address offset that ends with at least *n* zero bits.

9.5.5. Discussion

The alignment-attribute is a nontransportable feature, is not required for most purposes, and should only be used with a thorough knowledge of the target system's storage organization and accessing mechanisms.

A data segment declared as OWN or GLOBAL is allocated in the appropriate OWN or GLOBAL program section. Its location is defined in terms of an *address offset*, that is, an address relative to the beginning of the program section. In BLISS–16 and BLISS–32, any address constitutes the boundary of one or more allocation units: Thus all addresses are byte boundaries, every other address (relative to zero) is a word boundary as well, and in BLISS–32 every fourth address is also a longword boundary, and so on.

By default, a data segment is allocated at an address offset that is "natural" for either its size or type; for example, a word-size scalar is aligned to a word boundary, and a structured segment is always fullword aligned, whatever its allocation unit.

In BLISS–16, where the value of boundary may be 0 or 1, the only meaningful use of the alignment-attribute is to force byte-size scalar items to a word boundary, presumably for reasons of execution efficiency in special situations.

In BLISS–32 the boundary value for OWN and GLOBAL data segments is limited only by physical-storage considerations. Further, the alignment-attribute can be used to specify a smaller as well as a larger boundary than the default (except for byte items, obviously), essentially for purposes of storage compaction versus execution efficiency.

A data segment declared in a LOCAL or STACKLOCAL declaration is allocated in the current stackframe. The stack handling mechanism imposes certain restrictions such that the alignment specified for a LOCAL or STACKLOCAL data segment cannot exceed a longword boundary in BLISS–32.

An example of the use of an alignment-attribute in BLISS–32 follows:

```
OWN
    X: ALIGN(3);
```

In this example the alignment-attribute, ALIGN(3), directs the compiler to allocate data-segment X in such a way that its binary address offset ends in at least three 0s. That is to say, it directs the compiler to align the segment to a quadword boundary. Depending on where available storage begins, the compiler must leave from zero to seven bytes of unused storage in order to satisfy this alignment attribute.

9.6. The Initial-Attribute

An initial-attribute can be used in an OWN, LOCAL, STACKLOCAL, REGISTER, GLOBAL-REGISTER, EXTERNAL-REGISTER, or GLOBAL data-declaration.

An initial-attribute supplies one or more initialization values, which are assigned to the data segment before program execution begins.

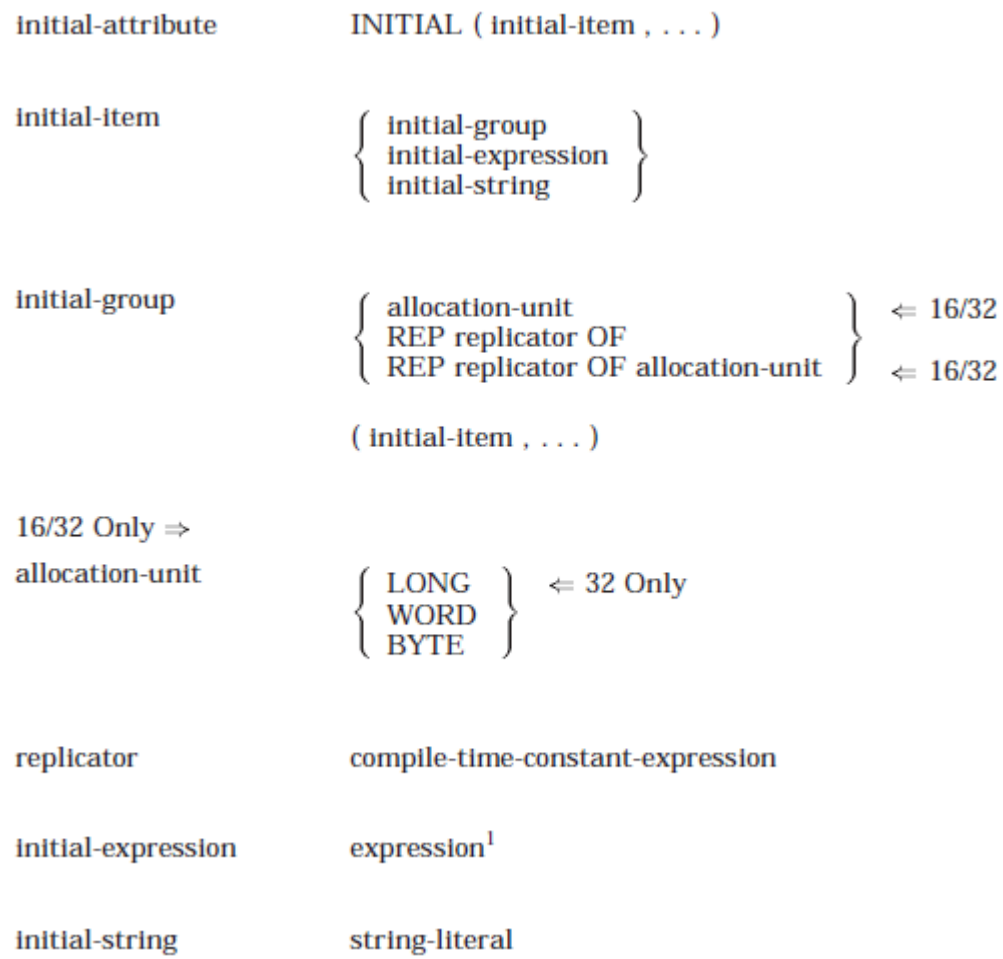
Examples of the use of initial-attributes follow:

```
OWN X: INITIAL(2);                  !X is initialized to 2.

GLOBAL Y: VECTOR[6]                 !Each element of Y is initialized
    INITIAL(REP 6 OF (-1));         !to -1.
```

16/32 Only:

```
GLOBAL Z: VECTOR[20,BYTE]           !The first 4 bytes of Z are
    INITIAL(BYTE('STOP',            !initialized to S, T, O, and
                 REP 16 OF (0)));   !P; the last 16 bytes to 0.
```

9.6.1. Syntax



1. The initial-item can be an executable expression, but is restricted in use to a link-time constant expression for OWN and GLOBAL declarations. For LOCAL, STACKLOCAL, REGISTER, GLOBAL REGISTER, and EXTERNAL REGISTER declarations the initial-item can be an executable expression.

9.6.2. Restriction

The initial-item values must not occupy more storage than is allocated for the data segment.

If a declaration contains both a structure-attribute and an initial-attribute, then the structure-attribute must precede the initial-attribute.

If a declaration contains both an allocation-unit (used as an attribute) and an initial-attribute, then the allocation-unit must precede the initial-attribute (BLISS–16/32 only).

9.6.3. Default

BLISS–16/32 ONLY

If an initial-attribute appears in the declaration of a scalar name without a structure-attribute being present, the default allocation-unit for the initial-items in the initial-attribute is the allocation-unit of the scalar name. Otherwise (without a structure-attribute), the default allocation-unit is WORD for BLISS–16 and LONG for BLISS–32.

9.6.4. Semantics

With the exception of the case where a LOCAL declaration is handling a non-PLIT item, the list of initial-items is evaluated as it would be in a PLIT. The resulting value is placed in the data segment at the time it is allocated. If the initial-item occupies less storage than the data segment, the trailing bits of the data segment are initialized to zeros.

9.6.5. Pragmatics

The use of the INITIAL attribute is the preferred method for initializing scalar data segments, while the use of the PRESET attribute (as described in *Section 9.7, "The Preset-Attribute"*) is the best method for initializing structured storage.

9.7. The Preset-Attribute

A preset-attribute can be used in an OWN, LOCAL, STACKLOCAL, REGISTER, GLOBAL-REGISTER, EXTERNAL-REGISTER, or GLOBAL data-declaration that declares a structured data-segment. It allows static initialization of individual fields of a structured data-segment.

A preset-attribute supplies an initialization value for one or more fields of a data structure, one value per specified field. These values are assigned to the data segment before program execution begins. Unspecified portions of the data segment are set to zero.

An example of the use of PRESET is given in the following program fragment, involving a block structure defined with field-names:

```
FIELD LINK_LIST_ITEMS =
     SET
     LL_VALUE = [0,0,%BPVAL/2,0],
     LL_TYPE = [0,%BPVAL/2,%BPVAL/2,0],
     LL_LAST = [1,0,%BPVAL,0],
     LL_NEXT = [2,0,%BPVAL,0]
     TES;

GLOBAL LLIST_HEAD : BLOCK[3] FIELD(LINK_LIST_ITEMS)
          PRESET( [LL_NEXT]  = LLIST_HEAD,
                  [LL_LAST]  = LLIST_HEAD,
                  [LL_VALUE] = -1 ) ;
```

In this example the origin block of a linked list is initialized with suitable values; note that the list of preset values is order independent. The LL_TYPE field is set to zero by default. The predeclared literal %BPVAL used in the example is defined in *Section 14.1.5, "Predeclared Literals"*.

9.7.1. Syntax

preset-attribute	PRESET (preset-item , . . .)
preset-item	[ctce-access-actual , . . .] = preset-value
ctce-access-actual	{ compile-time-constant-expression / field-name }
preset-value	expression[1]

1. For OWN and GLOBAL declarations the preset-value must be a link-time constant expression. For LOCAL, STACKLOCAL, REGISTER, GLOBAL REGISTER, and EXTERNAL REGISTER declarations the preset-value can be an executable expression.

The field-name is defined in *Chapter 11, "Data Structures"*.

9.7.2. Restriction

Within the declaration (OWN, LOCAL, and so forth), the preset-attribute must be preceded by a structure-attribute.

If any preset-item contains a field-name, the preset-attribute must be preceded by a field-attribute designating that field-name.

The preset-attribute and initial-attribute cannot be used in the same declaration. A declaration cannot contain more than one preset-attribute.

The preset values must not occupy more storage than is allocated for the data segment, and the fields described by the preset-items cannot overlap.

When expanded, the structure-reference formed by concatenating the declaration name with the bracketed access-actual list of a preset-item must only yield a link-time constant expression for an OWN or GLOBAL declaration. The value of that expression must be within the range of addresses allocated to the data-segment. Also, if that expression is a field-reference, it must conform to the dialect-specific restrictions on field-references used in an assignment context, as specified in *Section 11.2, "Field-References"*.

9.7.3. Default

When a preset-attribute appears in one of the declarations, any portion of the segment not described by a preset-item is set to zeros on allocation.

9.7.4. Semantics

The declaration name (OWN, LOCAL, and so forth) is concatenated with each preset-item, in turn, and the expressions so formed are evaluated as if they were assignment expressions. The resulting values are placed in the data segment at the time it is allocated. Any portions of the data-segment not explicitly initialized by preset-items are set to zeros.

9.7.5. Pragmatics

The use of the PRESET attribute is the preferred method for initializing nonscalar data-segments, although some simple VECTOR-type structures can be initialized conveniently with the INITIAL attribute. Initialization of most heterogeneous structures with the INITIAL attribute is, however, impractical or at least an error-prone practice.

Note

A psect-allocation attribute can be used to conveniently assign an initialized data-segment to write-protected storage. See *Section 9.8, "The Psect-Allocation Attribute"* for more information.

Assignment-expressions involving a structure-reference as their left operand are, in effect, evaluated during the initialization process and must meet the following conditions:

1. Must be resolvable at link time for an OWN or GLOBAL declaration.

2. Must result only in stores to locations allocated to the named data-segment (with no spillover).

3. Must result in assignments that are valid for the intended target systems, in terms of field size and word-boundary constraints (if any). For example, in all dialects a field to be stored into (or fetched from) may not be longer than a fullword.

The specific restrictions on field-references (the typical result of structure-reference expansions) are fully described in *Chapter 11, "Data Structures"*.

These restrictions come into play only in the case of a relatively complicated structure, such as one whose definition contains a routine call or performs bounds checking, for example. They pose no problem for the initialization of predeclared structures and other comparably straightforward user-declared structures.

9.8. The Psect-Allocation Attribute

The psect-allocation attribute can be used in declarations of permanent data-segments and in declarations of routines. It specifies the name of the program section in which the declared data-segment or routine (code segment) is to be allocated. Program sections and the psect-declaration are described in *Chapter 18, "Special Features"*.

The psect-allocation attribute provides a more convenient means of making program-section assignments for OWN, GLOBAL, and code segments than is possible using the psect-declaration alone. A major use of the psect-allocation attribute is for assigning an OWN or GLOBAL data-segment to write-protected storage. For example:

```
GLOBAL LITERAL
    MAIN_POWER = 0, AUX_POWER = 1, PRIMARY_BYPASS = 2,
    VALVE_1 = 3, VALVE_2 = 4, SECOND_BYPASS = 5, DUMPER = 6,
    OFF = 0, ON = 1 ;

GLOBAL  STARTUP_STATE : BITVECTOR[7]  PSECT( $PLIT$ )
           PRESET([MAIN_POWER]     = ON ,
                  [AUX_POWER]      = OFF ,
                  [VALVE_1]        = ON ,
                  [VALVE_2]        = OFF ,
                  [PRIMARY_BYPASS] = OFF ,
```

```
[SECOND_BYPASS]   = ON ,
[DUMPER]          = OFF ) ;
```

This fragment of a supposed process-control program establishes a control table of symbolically named binary values for use by several modules and, because its content should never be modified, it is allocated in the $PLIT$ program-section, by means of the PSECT attribute. $PLIT$ names the default program section for PLIT storage, which is given read-only access protection (if available on a given target system).

9.8.1. Syntax

psect-allocation	PSECT (psect-name)
psect-name	name

9.8.2. Restrictions

The psect-allocation attribute can appear in the following data- and routine-declarations only: FORWARD, OWN, GLOBAL, EXTERNAL, FORWARD ROUTINE, ROUTINE, GLOBAL ROUTINE, EXTERNAL ROUTINE.

The psect-name specified in the attribute must either be a predeclared, default program-section name or be explicitly declared in a psect-declaration before its use (see *Section 18.1, "Psect-Declarations"*). If specified in a FORWARD or FORWARD ROUTINE declaration, the psect-name must match the psect-name explicitly or implicitly associated with the controlling declaration of the data-segment or routine.

9.8.3. Defaults

If no psect-allocation attribute is specified, then the declared data- or code-segment is allocated in the program section established by the most recent psect-declaration for the segment's storage class (OWN, GLOBAL, or CODE), or in the appropriate default program section.

9.8.4. Semantics

In declarations other than EXTERNAL or EXTERNAL ROUTINE, the psect-allocation attribute causes the declared data-segment or code-segment to be allocated in the named program section.

In EXTERNAL and EXTERNAL ROUTINE declarations, the psect-allocation attribute informs the compiler that the declared segment is allocated in the named program section of another module (presumably), and any attributes defined for that program section in the current module are to apply.

9.8.5. Pragmatics

While the psect-allocation attribute need not appear in a FORWARD or FORWARD ROUTINE declaration, its specification in those declarations can favorably affect the quality of code generated for the segment in question, particularly in the case of FORWARD ROUTINE. Note that there is no default program-section name associated with a FORWARD or FORWARD ROUTINE declaration.

The psect-allocation attribute is essentially a convenience, allowing you to more easily achieve what would otherwise require repeated uses of the PSECT declaration.

9.9. The Volatile-Attribute

A volatile-attribute can be used in any data-declaration other than a REGISTER declaration. It can also be used in a bind-data-declaration.

For purposes of optimization, the compiler assumes that the contents of a data segment will be changed during execution in either of two ways: by an assignment or by a routine-call. The volatile-attribute specifies that the contents of the declared data segment can change in a third way: by an action that is not directly specified in the module being compiled. This attribute causes the compiler to assume that the value in the declared data segment can change at any time. Consequently the compiled code must fetch the contents of that data segment anew for each fetch in the BLISS program and must store a value for each assignment.

An example of the use of a volatile-attribute follows:

```
GLOBAL INPUT_PORT: VOLATILE;
```

In this example, it is assumed that INPUT_PORT designates a data segment that is set, through an interrupt routine, whenever a fullword of input arrives.

9.9.1. Syntax

volatile-attribute VOLATILE

9.9.2. Semantics

A volatile attribute is a warning to the compiler that the contents of a data segment can change at any time. A module that does not declare each such data segment as VOLATILE is invalid.

If the volatile-attribute appears in the declaration of the name of a REF structure (as described in *Section 11.1.3.5, "REF Structures"* and *Section 11.4, "Structure-Attributes and Storage Allocation"*), then the volatile attribute applies both to the storage for the address of the structure and to the storage for the structure itself.

9.10. The Novalue-Attribute

The novalue-attribute can be used in a routine-declaration or a bind-routine-declaration. It specifies that the declared routine does not return a value.

It is usually possible to determine by inspection whether or not a routine returns a value. However, in order to facilitate optimization and to provide clear documentation, this information must be given as part of the declaration of the routine-name. Specifically, the novalue-attribute must or must not be used depending on whether the routine does not or does return a value.

An example of a routine that does not return a value follows:

```
ROUTINE EXCH(X,Y): NOVALUE =    !There is a NOVALUE attribute, so the
    BEGIN                       !routine does not return a value;
    LOCAL TEMP;                 !instead, its effect is to exchange
    TEMP = ..X;                 !the values of X and Y.
```

```
.X = ..Y;
.Y = .TEMP;
END;
```

This routine, having no RETURN expression, returns control after complete evaluation of the routine-body. Because the routine-body is a block that consists solely of block-actions (expressions terminated by a semicolon) and has no block-value, no value is returned. The NOVALUE attribute affirms this procedure-like characteristic. See *Section 8.1, "Blocks"* for a discussion of block-actions and block-values.

Note that if routine EXCH did not contain the NOVALUE attribute, the compiler would assume that a null expression (namely the block-value expression) exists between the last expression shown and the block terminator. This in turn would cause the compilation diagnostic "Null expression appears in value-required context". When such a routine is called, it may appear to return a value, but that value is unpredictable.

Alternatively, if the last assignment expression were not terminated by a semicolon (and NOVALUE was specified), the routine would indeed have a block-value – the value of that assignment expression. However, that value would be discarded prior to return of control because of the NOVALUE attribute. Thus, a routine with the NOVALUE attribute never has a return value, no matter what value-implying expressions appear in its body.

9.10.1. Syntax

novalue-attribute NOVALUE

9.10.2. Restrictions

A routine that is declared with a novalue-attribute must not be called in a context that requires a value.

9.10.3. Semantics

The value of a routine that is declared with the novalue-attribute is undefined.

9.11. The Linkage-Attribute

The linkage-attribute can be used in a routine-declaration or a bind-routine-declaration. It specifies a *linkage-name* that is associated with the declared routine-name. This, in turn, causes the routine-name to be associated with the linkage-declaration that governs that linkage-name. The linkage-definition identified by the linkage-name controls both the code generated for the given routine and the code generated for any call to that routine.

A *linkage* is the mechanism used to call a routine; it saves registers, passes parameters, and controls other aspects of communication between a routine-call and the called routine. The default linkage-name BLISS in BLISS–16/32, or BLISS36C in BLISS–36, identifies the standard linkage convention for BLISS–compiled routines.

The linkage-attribute is simply a name; it is the declaration of that name that specifies the linkage to be used. BLISS includes several predeclared linkage-names. Linkage-declarations and predeclared linkage-names are described in *Chapter 13, "Linkages"*.

9.11.1. Syntax

linkage-attribute	linkage-name
linkage-name	name

9.11.2. Restrictions

A linkage-name must be one of the predeclared linkage-names or must be governed by a linkage-declaration.

A linkage-attribute given for a routine-name in an EXTERNAL ROUTINE, FORWARD ROUTINE, BIND ROUTINE, or GLOBAL BIND ROUTINE declaration must be the same as the linkage-attribute given in the corresponding ROUTINE or GLOBAL ROUTINE declaration.

9.11.3. Defaults

The default linkage-attribute is the predeclared linkage-name BLISS for BLISS– 16 or BLISS–32, and the linkage-name BLISS36C for BLISS–36.

9.11.4. Semantics

A linkage-attribute associates a linkage-name with a routine-name. Thus, the linkage-attribute indirectly controls the linkage-related code generated for a ROUTINE or GLOBAL ROUTINE DECLARATION, and the code generated for all calls to the routine, according to the definition of the specified linkage-name.

9.12. The Range-Attribute

The range-attribute can be used in a literal-declaration or external-literal-declaration. These declarations are described in *Chapter 14, "Binding"*.

A literal-name designates a constant value that is used as data but is stored in the object code rather than in a data segment. When the compiler is provided with sufficient information and the literal value is small enough, a short field can be generated for the value rather than a fullword.

The range-attribute specifies the quantity of storage required for a literal and indicates whether the field is to be interpreted as a signed or unsigned representation.

An example of the use of the range-attribute follows:

```
EXTERNAL LITERAL X: UNSIGNED(4);
```

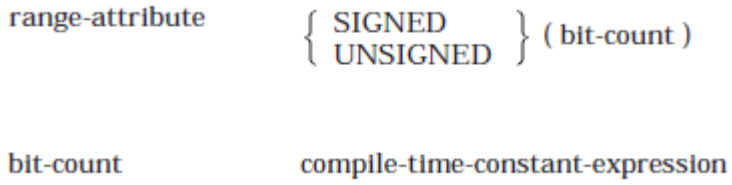
The effect of this attribute in a BLISS–32 context is as follows. Analogous effects would be obtained on other target systems. At the time the module containing this declaration is compiled, it is assumed that the value of X can be accommodated in a literal-operand specifier, and code is generated on that assumption. Then, when the modules are linked, a check is made for agreement of the range-attribute with the external value and the value of X is then placed in the empty fields provided for it.

Suppose the following declaration appears in another module of the same program:

```
GLOBAL LITERAL X = 12: UNSIGNED(4);
```

This declaration not only specifies that X designates the value 12, but also that it can be stored as an unsigned integer in four bits. This attribute both documents that a range-attribute assumption exists in another module of the program and allows the compiler to verify that the assumption is satisfied.

9.12.1. Syntax



9.12.2. Restriction

The value, *n*, of bit-count must be in the range $1 \leq n \leq$ %BPVAL. That is, the field specified cannot be longer than a fullword.

9.12.3. Default

The default range-attribute is SIGNED(%BPVAL).

9.12.4. Semantics

The range-attribute specifies the maximum number of bits required for a given literal value, and indicates whether the value is to be interpreted as a signed or unsigned integer.

9.13. The Addressing-Mode-Attribute – BLISS–16/32 Only

Each data or routine name has, as its value, an address. As the compiler translates a BLISS module into an object module, it replaces each use of a data or routine name with an offset address value. The final address value is supplied later by the linker and the operating system. But the compiler does provide a sequence of bytes in the object code to accommodate the final address value.

An address can be encoded as either absolute or relative, and in either a short or long form, and a PDP–11 address can be encoded as either absolute or relative. The addressing-mode-attribute determines the way in which the address is encoded. For every use of a data or routine name, the default rules specify an addressing-mode-attribute (if one is not given explicitly).

An addressing-mode-attribute can be given in an OWN, GLOBAL, FORWARD, or EXTERNAL declaration (described in *Chapter 10, "Data Declarations"*), or in a ROUTINE, GLOBAL ROUTINE, FORWARD ROUTINE, or EXTERNAL ROUTINE declaration (described in *Chapter 12, "Routines"*). This attribute can also be used in a PSECT declaration (*Section 18.1, "Psect-Declarations"*), and in a SWITCHES declaration or a module-head switch (*Section 18.2, "Switches-Declarations"* and *Section 19.2, "Module-Switches"* respectively). The latter two uses indirectly control a number of individual data and routine declarations.

9.13.1. Syntax

16/32 Only ⇒

addressing-mode-attribute — ADDRESSING_MODE { mode-16 | mode-32 }

mode-16 — { ABSOLUTE | RELATIVE }

mode-32 — { GENERAL | ABSOLUTE | LONG_RELATIVE | WORD_RELATIVE }

9.13.2. Default

Consider a name that is declared by one of the following declarations:

own-declaration
global-declaration
forward-declaration
external-declaration
routine-declaration
global-routine-declaration
forward-routine-declaration
external-routine-declaration
psect-declaration

For a name so declared, the addressing-mode-attribute is obtained by the following rules:

- If a default PSECT is associated with one of these declarations, the mode declared in the psect is used. Thus, OWN, GLOBAL, and ROUTINE declarations would use psect addressing modes of OWN, GLOBAL, and CODE, respectively (as described in *Section 18.1, "Psect-Declarations"*).
- If the declaration type is FORWARD or FORWARD ROUTINE, the mode established by the ADDRESSING_MODE (NONEXTERNAL#= . . .) module-head switch or the switches declaration is used (as described in *Section 18.2, "Switches-Declarations"* and *Section 19.2, "Module-Switches"*).
- If the declaration type is EXTERNAL or EXTERNAL ROUTINE, the mode established by the ADDRESSING_MODE (EXTERNAL#= . . .) module-head switch or the switches declaration is used (as described in *Section 18.2, "Switches-Declarations"* and *Section 19.2, "Module-Switches"*).

If a PSECT attribute is given, the addressing mode specified in the psect is used as shown in the following example:

```
OWN
    X: PSECT(GEN)
    ADDRESSING_MODE( WORD RELATIVE );
```

If an ADDRESSING_MODE attribute is given, the addressing mode specified by the switch is used. If both PSECT and ADDRESSING_MODE are used, then the last attribute encountered determines the addressing mode.

9.13.3. Semantics

The compiler translates each use of a data or routine name into an *encoded address*. An encoded address consists of an *encoding-type* followed by a *displacement*. The encoding-type specifies the addressing-mode-attribute and other information, while the displacement is an address specification. The encoding-type always occupies one byte, while the displacement occupies a number of bytes that is determined by the addressing-mode-attribute. The addressing-mode-attribute instructs the compiler in the preparation of an encoded address, as follows:

Attribute	**Instruction to Compiler**
GENERAL	Let the linker make the choice between using a relative displacement or an absolute value. Provide four bytes for the displacement, or value, and one byte for the addressing mode descriptor.
ABSOLUTE	Use an absolute value. If BLISS–32, put in four bytes. If BLISS–16, put in two bytes.
LONG_RELATIVE	Use a relative displacement, and put it in four bytes.
WORD_RELATIVE	Use a relative displacement, and put it in two bytes.
RELATIVE	Use a relative displacement, and put it in two bytes.

The RELATIVE and WORD_RELATIVE attributes apply to most names (each is the ultimate default for its mode), and are appropriate for references within executable images that are not unusually large. The LONG_RELATIVE attribute is used in the infrequent situation where 16 bits is not sufficient to represent a relative address. The ABSOLUTE attribute is used for names that designate addresses that are fixed in the address space, such as system service routines, device register addresses, and data. The GENERAL attribute is used when the choice between an absolute or relative address cannot be made at compile time.

9.14. The Weak-Attribute – BLISS–32 Only

The weak-attribute can be used in a declaration that has either GLOBAL or EXTERNAL in its keyword phrase. Such declarations are described in the following chapters.

The weak-attribute affects the way in which the VMS Linker and VMS Librarian programs handle global names. This is discussed further under EXTERNAL declarations, in *Section 10.4.3, "Semantics"*.

9.14.1. Syntax

```
32 Only ⇒
weak-attribute          WEAK
```

9.14.2. Semantics

The weak-attribute specifies a property of a name for use by the linker and librarian programs, as described in the manuals for those programs.

9.15. A Summary of Attribute Usage

Each attribute description in this chapter includes a list of the declarations in which the attribute can be used. That information is gathered together in the following table, where an "x" marks each attribute that can be used in each kind of declaration.

	Allocation-Unit	Extension	Structure	Field	Alignment	Initial	Preset	Psect-Allocation	Volatile	Novalue	Linkage	Range	Addressing-Mode	Weak
OWN	X	X	X	X	X	X	X	X	X	.	.	.	X	.
GLOBAL	X	X	X	X	X	X	X	X	X	.	.	.	X	X
FORWARD	X	X	X	X	.	.	.	X	X	.	.	.	X	.
EXTERNAL	X	X	X	X	.	.	.	X	X	.	.	.	X	X
LOCAL	X	X	X	X	X	X	X	.	X	.	.	.	.	.
STACKLOCAL	X	X	X	X	X	X	X	.	X	.	.	.	.	.
REGISTER	X	X	X	X	.	X	X	.	.	.	.	.	.	.
GLOBAL REG.	X	X	X	X	.	X	X	.	.	.	.	.	.	.
EXTERNAL REG.	X	X	X	X	.	X	X	.	.	.	.	.	.	.
MAP	X	X	X	X	.	.	.	.	X	.	.	.	.	.
BIND	X	X	X	X	.	.	.	.	X	.	.	.	.	.
GLOBAL BIND	X	X	X	X	.	.	.	.	X	.	.	.	.	X
ROUTINE	.	.	.	.	.	.	.	X	.	X	X	.	X	.
GLOBAL RTN	.	.	.	.	.	.	.	X	.	X	X	.	X	X
FORWARD RTN	.	.	.	.	.	.	.	X	.	X	X	.	X	.
EXTERNAL RTN	.	.	.	.	.	.	.	X	.	X	X	.	X	X
BIND ROUTINE	.	.	.	.	.	.	.	.	.	X	X	.	.	.
GLOBAL BIND RTN	.	.	.	.	.	.	.	.	.	X	X	.	.	X
LITERAL	.	.	.	.	.	.	.	.	.	.	.	X	.	.
GLOBAL LIT	.	.	.	.	.	.	.	.	.	.	.	X	.	X
EXTERNAL LIT	.	.	.	.	.	.	.	.	.	.	.	X	.	X

Chapter 10. Data Declarations

A data-declaration describes one or more data segments. Taken together, the data declarations of a program specify the storage required for the data on which that program operates.

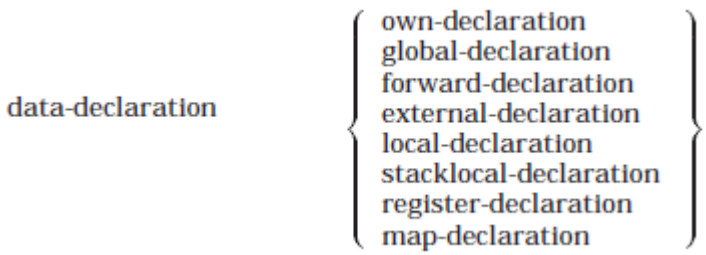
The data-declarations can be divided into three categories, as follows:

- A *permanent* declaration begins with OWN, GLOBAL, or EXTERNAL. It describes a data segment that remains allocated throughout the execution of the program.
- A *temporary* declaration begins with LOCAL, STACKLOCAL, REGISTER, GLOBAL REGISTER, or EXTERNAL REGISTER. It describes a data segment that exists only during each execution of a given block.
- An *overlay* declaration begins with MAP. It describes a data segment that has been declared elsewhere, but that is given new attributes by this declaration.

A data-declaration provides some or all of the following information about each data segment it declares:

- The *name* of the data segment.
- The *address* of the data segment, which is determined by the kind of declaration and by some of the attributes. The address of the data segment becomes the value of the declared name.
- The *scope* of the name of the data segment, which depends on the position of the declaration within the program and on the kind of declaration.
- The *longevity* of the data segment, which is determined by the kind of declaration (permanent or temporary).
- The *attributes* of the data segment, which are given as part of the declaration and by the default rules for attributes.

The attributes applicable to data-declarations are described in *Chapter 9, "Attributes"*, except for the structure-attribute, which is described in *Chapter 11, "Data Structures"* along with other aspects of data structures. The syntax diagram for data-declarations is as follows:



10.1. Own-Declarations

The storage for an OWN data segment is permanent; that is, it is created before program execution begins and exists throughout program execution. The scope of an own-declaration is its immediately

containing block (including any lower-level blocks contained therein). That is to say, the name of an OWN data segment can be used only within the block in which it is declared.

An example of an own-declaration in a routine-declaration context follows:

```
ROUTINE KILO  =
    BEGIN
    OWN
        X:  INITIAL(0);
    X = .X+1;
    IF .X LEQ 1000 THEN 1 ELSE 0
    END;
```

The data segment named X is allocated and initialized only once, before program execution begins. It can be referred to by the name X only within the routine KILO.

10.1.1. Syntax

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

own-item own-name { : own-attribute . . . / nothing }

own-name name

own-attribute { allocation-unit ⇐ 16/32 Only; extension-attribute ⇐ 16/32 Only; structure-attribute; field-attribute; alignment-attribute ⇐ 16/32 Only; initial-attribute; preset-attribute; psect-allocation; volatile-attribute }

10.1.2. Restrictions

BLISS–16/32 Only

A structure-attribute must not appear in the same declaration as an allocation-unit or an extension-attribute. If the declaration contains both an allocation-unit attribute and an initial attribute, the allocation-unit must precede the initial-attribute.

A field-attribute can appear only in a declaration that has a structure-attribute.

If the declaration contains both a structure-attribute and an initial-attribute, the structure-attribute must precede the initial-attribute.

If the declaration contains both a structure-attribute and a preset-attribute, the structure-attribute must precede the preset-attribute.

An initial-and a preset-attribute must not appear together in the declaration.

The declaration must not contain more than one initial-or preset-attribute.

If the preset-attribute contains a field-name, the preset-attribute must be preceded by a field-attribute that designates the field-name.

10.1.3. Semantics

The data segment designated by a name that is declared OWN is allocated in the current program section for the storage class OWN, as described in *Section 18.1, "Psect-Declarations"*. Program sections for the storage class OWN are created before program execution begins and are not discarded until after program execution is complete.

The data segment for an OWN name is always allocated at the lowest possible address within the unused portion of the current OWN program section, after allowing for address-alignment requirements (if any).

In BLISS–16, data segments larger than one byte are allocated at even addresses, which may leave an unused byte preceding the data segment. One-byte data segments are allocated at the next available byte.

In BLISS–32 the address must be consistent with the alignment-attribute, which is either given explicitly or determined by default. The alignment-attribute may dictate some unused bytes, as described in *Section 9.5, "The Alignment-Attribute – BLISS–16/32 Only"*.

In BLISS–36 there are no special alignment rules; each data segment is allocated at the next available word.

Because OWN data segments are allocated in this way, the address of one OWN data segment can be calculated relative to that of another, provided that both segments are declared in the same module and allocated in the same program section.

When the storage for an OWN data segment is created by the linker, it is set to zeros. If the data segment is given an initial value in the declaration, it is initialized by the linker.

10.2. Global-Declarations

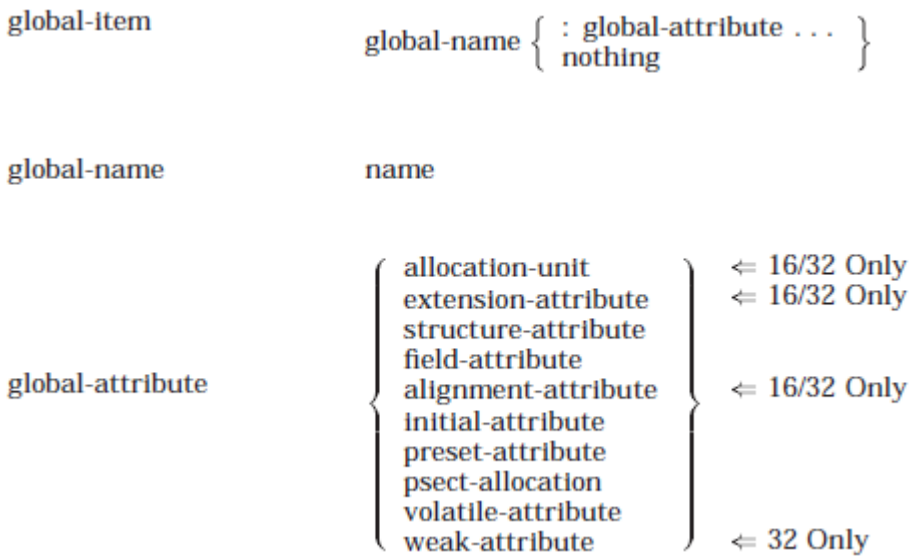
Like an OWN data segment, the storage for a GLOBAL data segment is permanent; that is, it exists throughout program execution. In contrast to an OWN data segment, the name of a GLOBAL data segment can be used in several separate blocks; that is, in the block in which it is declared GLOBAL and in each block in which it is declared EXTERNAL.

Usually the block in which a name is declared GLOBAL is in one module and the blocks in which it is declared EXTERNAL are in other modules. In this way, a data segment can be shared among several modules.

Aside from the initial keyword, the syntax of the own-declaration and global-declaration is identical, except that in BLISS–32 the weak-attribute is permitted in a global-declaration.

10.2.1. Syntax

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



10.2.2. Restrictions

A name is declared as *global* when the declaration begins with the keyword GLOBAL, except for GLOBAL REGISTER (see *Section 10.8, "Global-Register-Declarations"*). A name must not be declared as global more than once in a program.

All the attribute restrictions given in *Section 10.1.2, "Restrictions"* also apply to GLOBAL declarations.

10.2.3. Semantics

The data segment designated by a name that is declared GLOBAL is allocated in the current program section for the storage class GLOBAL, as described in *Section 18.1, "Psect-Declarations"*. Program sections for the storage class GLOBAL are created before program execution begins and are not discarded until after program execution is complete.

The data segment for a GLOBAL name is allocated in the same predictable way as the data segment for an OWN name. Therefore, a programmer can determine the relative addresses of any two GLOBAL data segments that are declared in the same module and are allocated in the same program section.

A GLOBAL data segment can be accessed by name within the scope of the declaration of its name. In addition, it can be accessed within the scope of any external-declaration of its name.

10.3. Forward-Declarations

A forward-declaration is used to give the attributes of a name before storage is allocated for the name. A forward-declaration is always used in conjunction with an own-declaration or a global-declaration; it is used to avoid what would otherwise be a circular definition of names.

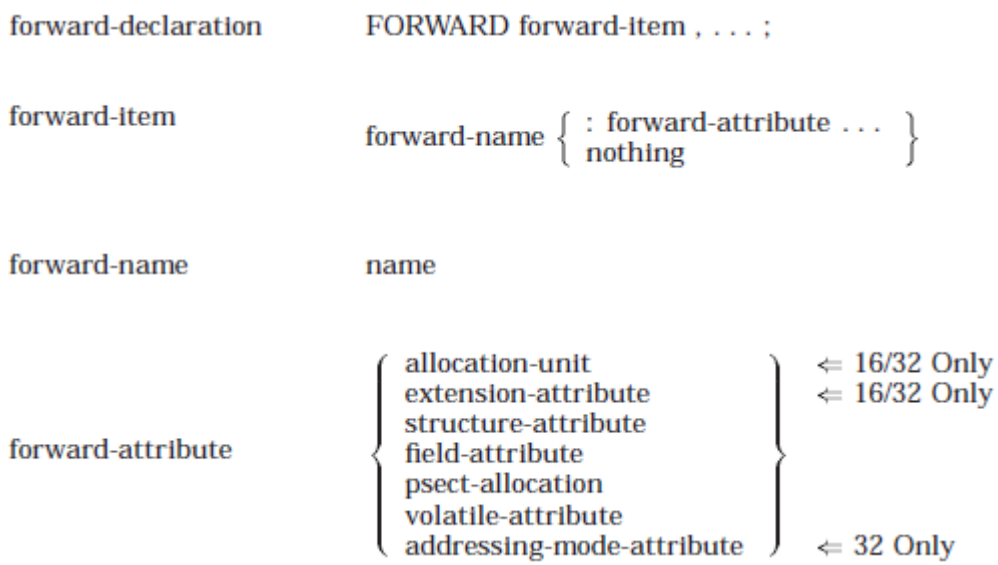
As an example, suppose that X and Y are pointers; that is, X and Y are each the name of a data segment that contains the address of another data segment. Suppose, also, that X and Y must be initialized to point to each other. The required declarations are as follows:

```
FORWARD
    Y;
```

```
OWN
    X:  INITIAL(Y),
    Y:  INITIAL(X);
```

The forward-declaration declares Y so that it can be used to initialize X which, in turn, is used to initialize Y.

10.3.1. Syntax



10.3.2. Restrictions

Each name that is declared by a forward-declaration must also be declared, a second time, by an own-declaration or a global-declaration that is in the same block.

After the default attributes have been filled in, a forward-declaration of a name and the associated own-declaration or global-declaration of the same name must be identical with respect to all the attributes allowed in the forward-declaration.

All the attribute restrictions given in *Section 10.1.2, "Restrictions"* also apply to FORWARD declarations.

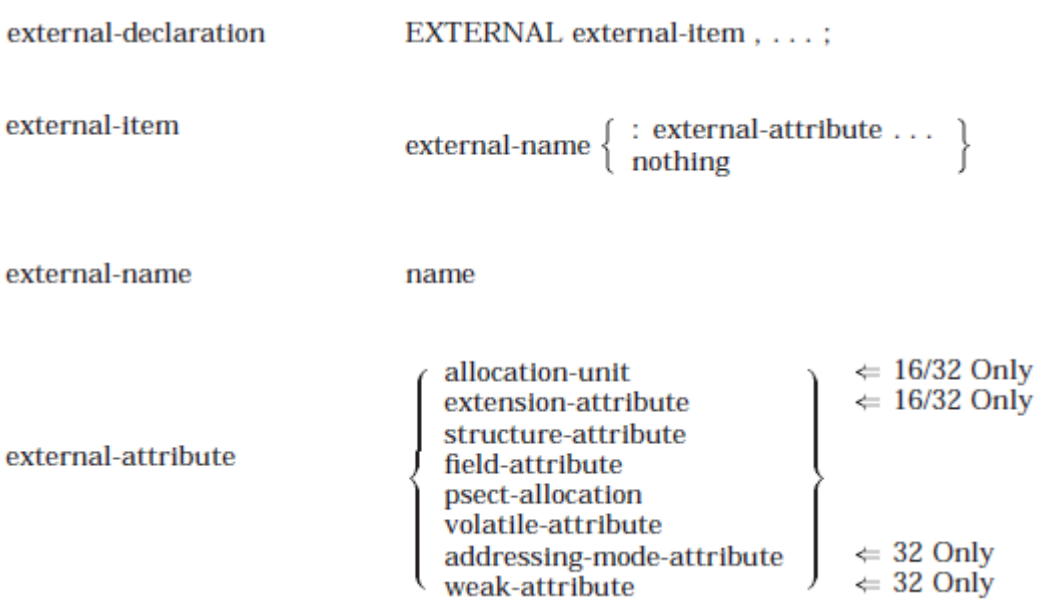
10.3.3. Semantics

The forward-declaration associates attributes with a name without allocating the storage for that name.

10.4. External-Declarations

A name that is declared EXTERNAL is assumed to be declared GLOBAL somewhere else in the same program. The linker treats each occurrence of the name governed by an external-declaration as if it were governed by the global-declaration of the same name. Thus the external declaration does not cause the allocation of a data segment but rather extends the accessibility of a data segment that is allocated elsewhere.

10.4.1. Syntax



10.4.2. Restrictions

A name that is declared EXTERNAL must also be declared GLOBAL somewhere else in the same program. In BLISS–32, this restriction does not apply if the EXTERNAL name has the weak-attribute.

All of the attribute restrictions given in *Section 10.1.2, "Restrictions"* also apply to EXTERNAL declarations.

After default attributes have been filled in, the following attributes of the EXTERNAL and GLOBAL declarations of a given name must be identical:

```
allocation-unit
extension-attribute
structure-attribute
field-attribute
volatile-attribute
```

10.4.3. Semantics

The linker generates and uses a list of all names that are declared GLOBAL in the entire program. For each such name, the list shows the value of the name and some of the attributes of the name. This list is used in determining the value of a given EXTERNAL name as follows:

- The list is searched for an entry for the given name. If such an entry is found, then it supplies the value of the given EXTERNAL name.
- In BLISS–32 only, if no entry for the given name is found and the given name has the weak-attribute, then zero is used as the value of the given name.
- If no entry for the given-name is found and the given name does not have the weak-attribute, then the program is not valid.

In BLISS–32 only, when an EXTERNAL name has the value zero (determined because no entry was found and the weak-attribute was present), the program can be executed provided an attempt is not made to use the given name as an address.

An EXTERNAL name already declared can be encountered in a GLOBAL or FORWARD declaration. If such a case arises during compilation, the following is done:

1. Parse the declaration.
2. Compare the attributes of the EXTERNAL declaration with those of the GLOBAL or FORWARD declaration.
3. If a mismatch occurs, generate a warning message.

10.5. Local-Declarations

The storage for a LOCAL data segment is temporary; that is, it exists only during the execution of the block in which it is declared. The data segment is allocated either in the stack frame for the block in which it is declared, or in a general register that is free.

The scope of a LOCAL data-declaration is its immediately containing block excluding any lower-level contained routines. That is, unlike OWN data segments, "up-level" references to a LOCAL data segment from a lower-level routine are not permitted.

10.5.1. Syntax

local-declaration	LOCAL local-item , . . . ;
local-item	local-name { : local-attribute . . . / nothing }
local-name	name
local-attribute	allocation-unit ⇐ 16/32 Only extension-attribute ⇐ 16/32 Only structure-attribute field-attribute alignment-attribute ⇐ 16/32 Only initial-attribute preset-attribute volatile-attribute

10.5.2. Restrictions

A local-declaration must be contained in a routine-body.

Suppose the routine-body of a given routine, routine A, contains the declaration of another routine, routine B. If a name is declared LOCAL in routine A and is not declared in routine B, then the name cannot be used in routine B. Such usage would be an "up-level" reference, which is prohibited for local-names.

A program must not depend on the relative positions of two LOCAL data segments in storage.

All of the attribute restrictions given in *Section 10.1.2, "Restrictions"* also apply to LOCAL declarations.

BLISS–32 only: An alignment-attribute used in the declaration of a LOCAL name must not have a boundary expression whose value is greater than 2.

10.5.3. Semantics

The data segment for a LOCAL name is allocated either in the current stack frame or in a general register. In either of the following situations, a given LOCAL data segment is always allocated in the current stack frame:

- The given data segment occupies more than a fullword.
- The name of the given data segment is used as an independent address; that is, its use is not confined to a fetch expression or to the left-hand side of an assignment expression.

In other situations, the choice between stack frame and register is based on strategies that the compiler uses for code optimization.

10.5.4. Pragmatics

A temporary data segment (such as a LOCAL data segment) must be used for a recursive variable in a recursive routine.

10.6. Stacklocal-Declarations

A STACKLOCAL data segment is always allocated in the current stack frame. In all other respects, it is the same as a LOCAL data segment.

10.6.1. Syntax

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

The local-item is as defined in *Section 10.5.1, "Syntax"*.

10.6.2. Restrictions

All of the attribute restrictions given in *Section 10.1.2, "Restrictions"*, and all the restrictions given in *Section 10.5.2, "Restrictions"* for LOCAL data segments, also apply to STACKLOCAL declarations.

10.6.3. Semantics

The semantics given in *Section 10.5.3, "Semantics"* for LOCAL data segments apply to STACKLOCAL data segments except that a STACKLOCAL data segment is always allocated in the current stack frame.

10.7. Register-Declarations

A register data segment is a data segment that is always allocated in a general register. In most other respects, it is the same as a LOCAL data segment. If the declaration specifies a register number, the data

segment is allocated in the specified register. Otherwise, the data segment is allocated in a register chosen by the compiler.

An example of a register-declaration follows:

```
REGISTER
    STATUS = 5: BITVECTOR[10],
    BETA;
```

This declaration associates the names `STATUS` and `BETA` with two general registers. The register number for `STATUS` is given explicitly as `5` and only `10` bits of that register are used. The register number for `BETA` is left to be chosen by the compiler, and the full register is used.

10.7.1. Syntax

register-declaration — REGISTER register-item , . . . ;

register-item — register-name

{ = register-number / nothing }

{ : register-attribute . . . / nothing }

register-name — name

register-number — compile-time-constant-expression

register-attribute — { allocation-unit ⇐ 16/32 Only / extension-attribute ⇐ 16/32 Only / structure-attribute / field-attribute / initial-attribute / preset-attribute }

10.7.2. Restrictions

The value of the register number, if specified, must be in the range given below for each dialect:

For BLISS–16:	0 through 5
For BLISS–32:	0 through 11
For BLISS–36:	0 through 12, if the governing linkage-attribute is BLISS36C (the default), FORTRAN_FUNC, or FORTRAN_SUB. 1 and 3 through 15, if the governing linkage-attribute is BLISS10. The general rule for BLISS–36 is that the register number must not specify a register in use as the stack pointer, the frame pointer, or the argument pointer

(if applicable). The linkage-definition that governs the routine containing the register-declaration controls the assignment of registers for these uses.

A register specified by register-number must be PRESERVED or NOTUSED in the linkage of any routine called in the containing block if the call occurs within the useful lifetime of the register data segment (that is, if the call occurs between the first and last possible references to that segment).

A register data segment must not occupy more than a fullword. A register-declaration must be contained in a routine-body.

Suppose the routine-body of a given routine, routine A, contains the declaration of another routine, routine B. If a name is declared REGISTER in routine A and is not declared in routine B, then the name cannot be used in routine B. Such usage would be an "up-level" reference and is not permitted for register data segments.

All the attribute restrictions given in *Section 10.1.2, "Restrictions"* also apply to REGISTER declarations.

A name declared in a register-declaration must be used only as the operand of a fetch expression or as the first operand of an assignment expression. This restriction does not apply to certain machine-specific-function parameters; see the applicable BLISS user manual.

10.7.3. Semantics

If a register-number is given in the declaration of a register data segment, then the data segment is allocated in that register. During execution of the routine that contains the declaration, the register can be used for other purposes, but none that conflict with the valid use of the allocated data segment.

A register data segment is similar to a local data segment in that it is created on entry to the block in which it is declared and released on exit from that block, and cannot be referenced from any lower-level contained routine-body.

10.7.4. Pragmatics

Standard register-names with appropriate predefined values are provided, as built-in names, for each BLISS dialect. In order to use these names with their predefined values, they may be declared in a BUILTIN declaration (*Section 18.3, "Built-In-Declarations"*). The built-in register-names and values are as follows:

BLISS–16		**BLISS–32**	
Name	**Value**	**Name**	**Value**
R0	0	R0	0
R1	1	R1	1
R2	2	R2	2
R3	3	.	.
R4	4	.	.
R5	5	.	.
SP	6	R11	11
PC	7	AP	12

		FP	13
		SP	14
		PC	15

BLISS–36 ONLY

The built-in register-names SP, FP, and AP are provided. The value defined for each name depends upon the linkage-definition associated with the routine in which the name is declared BUILTIN (see *Chapter 13, "Linkages"*).

10.8. Global-Register-Declarations

A *global register data segment* is a data segment that is created and allocated in a given register in one routine, and can be made available for use in other routines that are called by the declaring routine. Global register data segments are identified by name, and both the calling and called routine must agree (through a matching set of register- and linkage-declarations) that a particular global register data segment is available.

A global register data segment is the same as an ordinary register data segment with respect to its use within the declaring routine.

A GLOBAL REGISTER declaration establishes the name and actual register assignment of a global register data segment and creates the storage (that is, allocates the register). For the data segment to be available to a called routine, that routine must specify the same name in an EXTERNAL REGISTER declaration and must specify both the name and register-number in the GLOBAL linkage-option of its governing linkage-definition.

10.8.1. Syntax

```
global-register-declaration     GLOBAL REGISTER register-item , . . . ;

register-item                   register-name
                                    = register-number
                                  { : register-attribute . . . }
                                  { nothing                     }

register-name                   name

register-number                 compile-time-constant-expression

                                { allocation-unit      }  <= 16/32 Only
                                { extension-attribute  }  <= 16/32 Only
register-attribute              { structure-attribute  }
                                { field-attribute      }
                                { initial-attribute    }
                                { preset-attribute     }
```

10.8.2. Restrictions

The register-number is constrained by the containing routine's linkage as described for ordinary register data segments in the first paragraph of *Section 10.7.2, "Restrictions"*, but is also constrained by the linkage-definition governing any called routine that refers to the declared global register data segment. The inter-routine requirements are described in *Chapter 13, "Linkages"*.

A register data segment must not occupy more than a fullword. A global-register-declaration must be contained in a routine-body.

Suppose the routine-body of a given routine, routine A, contains the declaration of another routine, routine B. If a name is declared GLOBAL REGISTER in routine A and is not declared in routine B, then the name cannot be used in routine B. Such usage would be an "up-level" reference and is not permitted for register data segments.

All the attribute restrictions given in *Section 10.1.2, "Restrictions"* also apply to GLOBAL-REGISTER declarations.

A name declared in a global-register-declaration must be used only as the operand of a fetch expression or as the first operand of an assignment expression. This restriction does not apply to certain machine-specific-function parameters; see the applicable BLISS user manual.

If the linkage definition of a called routine specifies a global register data segment, then the routine call must be in the scope of a global- or external-register-declaration of the data segment.

BLISS–16/36 ONLY

If a call to a routine occurs in the scope of a global register data segment, then the register number of the data segment must be given in either the GLOBAL or PRESERVE linkage-option of the called routine's linkage definition.

BLISS–32 ONLY

If a call to a routine with CALL linkage-type occurs in the scope of a global register data segment, then the register number of the data segment must be given in either the GLOBAL or PRESERVE linkage-option of the called routine's linkage definition.

If a call to a routine with JSB linkage-type occurs in the scope of a global register data segment, then the register-number of the data segment must be given in either the GLOBAL or NOTUSED linkage-option of the called routine's linkage definition.

10.8.3. Semantics

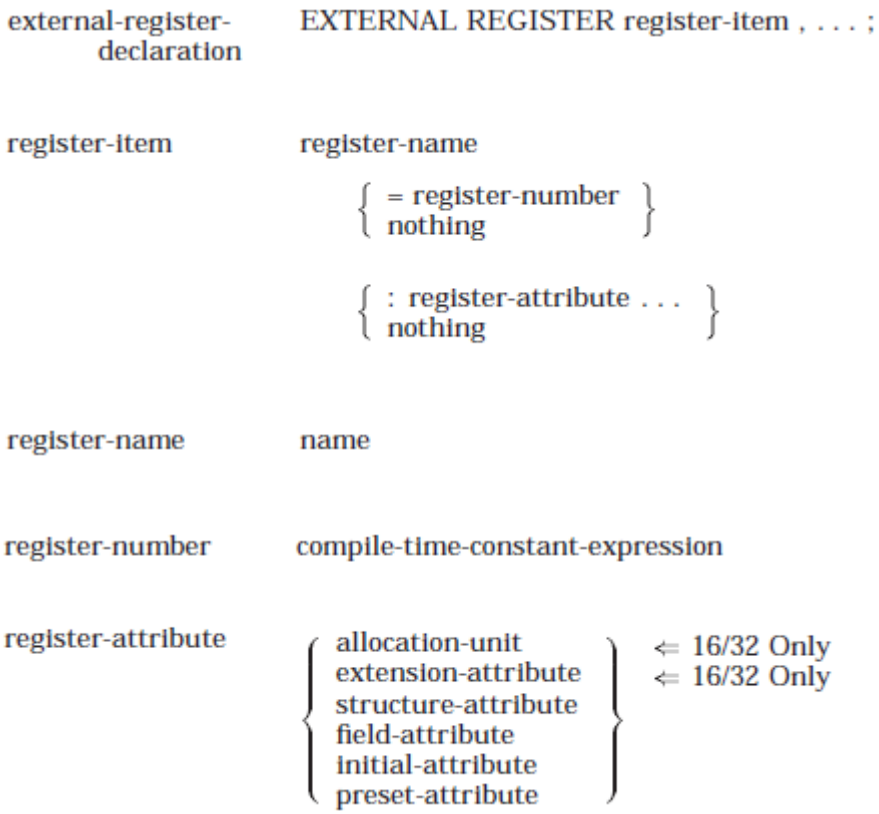
A global register-declaration causes a register data segment to be allocated. A global register data segment is a local data segment just like an ordinary register data segment – it is created on entry to the block in which it is contained and released on exit from that block. However, unlike an ordinary register data segment, a global register data segment is available in called routines under certain conditions, described briefly below and more fully in *Chapter 13, "Linkages"*.

In order to pass a global register data segment to a called routine, the linkage-definition of the called routine must contain the name and register-number of the data segment in its GLOBAL linkage-option. There may be more global register data segments available at a call than are specified in the linkage for the call; however, every global register data segment specified in the linkage must be available at the call. Only those global register data segments specified in the linkage are available in the called routine.

10.9. External-Register-Declarations

An EXTERNAL REGISTER declaration specifies that a global register data segment created in a calling routine is used in the routine containing the declaration. This declaration must be used in combination with linkage definitions that include appropriate GLOBAL linkage-options.

10.9.1. Syntax



10.9.2. Restrictions

The register number, if given, must be the same as that specified in the GLOBAL linkage-option of the containing routine's linkage definition. A register data segment must not occupy more than a fullword.

An external-register-declaration must be contained within a routine declaration whose linkage definition specifies the named global-register-segment.

Suppose the routine-body of a given routine, routine A, contains the declaration of another routine, routine B. If a name is declared EXTERNAL REGISTER in routine A and is not declared in routine B, then the name cannot be used in routine B. Such usage would be an "up-level" reference and is not permitted for register data segments.

All of the attribute restrictions given in *Section 10.1.2, "Restrictions"* also apply to external-register declarations.

A name declared in an external-register-declaration must be used only as the operand of a fetch expression or as the first operand of an assignment expression. This restriction does not apply to certain machine-specific-function parameters; see the applicable BLISS user manual.

10.9.3. Defaults

If an external-register-declaration does not specify a register number, the register-number given for that external-register-name in the GLOBAL linkage-option is assumed.

10.9.4. Semantics

An external-register-declaration specifies that a global register data segment created in a calling routine is available for use. The declared name must also be specified in the called routine's linkage definition; however, not all of the global register data segments specified in the linkage need be declared in an external-register-declaration.

BLISS–16/36 ONLY

If a global-register-segment is specified in the routine's linkage but is not declared EXTERNAL REGISTER, then the contents of the register are preserved by the called routine and the register is available for other purposes.

BLISS–32 ONLY

If a global-register-segment is specified in the routine's linkage but is not declared EXTERNAL REGISTER, then in a routine with CALL linkage-type the contents of the register are preserved by the called routine and the register is available for other purposes. In a routine with JSB linkage-type, however, the contents of such a register cannot be preserved and the register is not usable in any way.

10.10. Map-Declarations

A map-declaration is used to supply new attributes in the current block to a name that is already declared.

The most common use of a map-declaration is in the declaration of the formal-names of a routine-declaration. Each formal-name is considered to be declared as a fullword, unsigned scalar data segment in an imaginary block that surrounds the routine-body. When those attributes are not suitable, a MAP declaration is used to override these defaults. This use of a map-declaration is discussed in *Chapter 12, "Routines"*.

10.10.1. Syntax

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

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

map-name            name

map-attribute       { allocation-unit       } <= 16/32 Only
                    { extension-attribute   } <= 16/32 Only
                    { structure-attribute   }
                    { field-attribute       }
                    { volatile-attribute    }
```

10.10.2. Restrictions

A map-declaration must lie within the scope of another declaration of the same name. The latter declaration must be a data-declaration or a bind-data-declaration.

For BLISS–16/32 only, a structure-attribute must not appear in the same declaration as an allocation-unit or an extension-attribute.

A field-attribute can appear only in a declaration that has a structure-attribute.

10.10.3. Semantics

The declaration of a name as MAP changes neither the value of the name nor the contents of the data segment designated by the name. Instead, the storage whose address is given by the declared name is re-interpreted in accordance with the attributes given in the map-declaration.

Chapter 11. Data Structures

A *data structure* is the framework for a collection of values that are stored under a single name. Certain frequently used data structures are predefined in BLISS; they are the vector, the bit vector, the block, and the block vector. The use of these data structures is described in *Chapter 3, "BLISS Values and Data Representations"*.

This chapter describes the features of BLISS that permit you to go beyond the predefined data structures and design special data structures that fit a particular application.

The first section of this chapter discusses the concepts of data structures and provides a detailed example of a specific data structure.

The next section describes the field-reference, which is the fundamental BLISS mechanism for accessing an element of a data structure.

The next seven sections describe the features of BLISS that are used to define and use a data structure; they are structure-declarations, structure-attributes, field-declarations, field-attributes, ordinary-structure-references, default-structure-references, and general-structure-references.

The final two sections return to the description of specific data structures. One section gives the full definition of each of the BLISS predefined structures. The remaining section gives several examples of user-defined structures.

11.1. Introduction to Data Structures

The BLISS facilities for user-defined data structures have the following benefits:

- *Generality*. If a specific application requires a data structure that is different from any predefined data structure, you can define a new data structure that fills the need.
- *Flexibility*. If a specific application requires a different representation for an existing kind of data structure (for example, one that requires less space), you can provide a new data structure that provides the required representation.
- *Machine independence*. If a program must depend on the architecture of the computer in order to save space or execution time, that dependence can be localized and concealed within the appropriate data structure definition.
- *Checking*. If references must be checked for validity (for example, vector subscript in range), an appropriate check can be built into a user-defined structure definition.

The design for a new data structure has three parts: the abstract definition, the concrete representation, and the programmed description. The abstract definition and concrete representation are part of the design of a program; although they may be written down as part of the documentation, they are not a part of the BLISS program. On the other hand, the programmed description of a data structure is part of the BLISS program in which the structure is used.

This introductory discussion of data structures requires a specific example; therefore, a data structure called a "decimal digit array" is carried through each section of this discussion. The concrete representation and programmed description for the example structure is first worked out for the VAX

and BLISS–32. Further on, concrete representations and programmed descriptions are given for the PDP–11 and BLISS–16, and the DECsystem–10/20 and BLISS–36.

11.1.1. The Abstract Definition of Data Structures

An *abstract definition* of a data structure specifies the structure, content, and usage of a particular collection of data in terms of its application, not in terms of a particular computer implementation. Indeed, the definition is abstract only if it applies equally to all possible representations of the data.

The abstract definition of a decimal digit array might be as follows:

```
A decimal digit array is a compact storage representation of
a sequence of decimal digits that permits reasonably quick access to
 individual digits.
```

The decimal digit array is not a predefined structure in BLISS and it is not even an especially important structure. However, it is typical of the sort of data structure that can be readily defined.

The abstract definition of the decimal digit array establishes four characteristics of the desired structure:

1. The word "compact" asserts that the representation cannot waste space, presumably because there will be many decimal digit arrays or because some of them will have many elements.

2. The word "sequence", as well as the word "array" in the name of the structure, indicates that the elements of the structure are ordered.

3. The words "decimal digit" indicate that each element can have ten distinct values, and these values are associated with the characters "0", "1", and so on, through "9".

   Note

   This characteristic asserts that each element accommodates a range of 10 values (which requires somewhat less than four bits), not that each element accommodates a decimal digit character code (which would require seven or eight bits in ASCII).

   ---

4. The phrase "permits reasonably quick access to individual digits" provides important information about the usage of the data structure.

11.1.2. The Concrete Representation of Data Structures

The *concrete representation* of a data structure determines which bits of memory are occupied by the data and how these bits are interpreted. The design of the representation depends on the following considerations:

- The amount of storage available for the structure. If the structure is big, it should not contain a large proportion of unused storage.

- The amount of time available for access to the fields of the structure. If the structure is accessed frequently, each access should be fast.

- The effect of the representation on program development. If the elements must be accessed during debugging, that access should be convenient.

- Compatibility with other representations of the same data. If a commitment to a given representation has already been made, it may be necessary to accept that representation even if it is not optimal.

The design of a concrete representation is difficult, especially at the beginning of a project. The facilities of BLISS permit you to change concrete representations easily, even after the project is under way.

The possible representations for a data structure can be ranked according to time and space requirements. The ranking can begin with those that have compact storage but slow access and proceed to those that have fast access but excessive storage.

As an example, such a ranking for the decimal-digit-array data structure on the VAX target system would be as follows:

1. Because 32 bits can accommodate any 9-digit decimal number, the array can be stored nine digits to a fullword. In this representation, however, access to a single digit requires considerable computation (conversion of a 32-bit binary integer to a 9-digit decimal integer).

2. Because 4 bits can accommodate 10 distinct values, the array can be stored eight digits to a fullword. This representation requires a conversion to get from the element value to an ASCII character, but the conversion is a simple addition or OR operation.

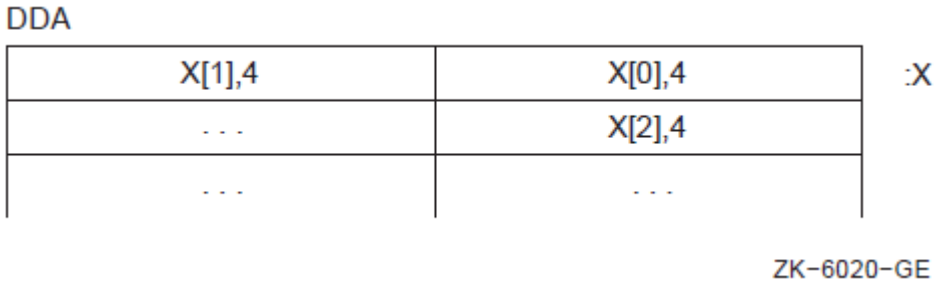
3. Because the ASCII codes for decimal digits normally occupy eight bits each, and because the byte is a natural unit of storage on the VAX, the array can be stored four digits per fullword. In this representation, about half the storage is wasted, but access is quicker.

4. Because the VAX works best on fullword values, the array could be stored one digit per fullword. This representation wastes a lot of storage, but provides the most rapid access.

Ranking representations in this way is useful, but sometimes difficult. Many considerations can affect the ranking, for example, both virtual and physical memory management strategies. The ranking might even be different for different models of the VAX.

Each of these concrete representations is correct for certain situations. For the example under consideration, the representation in item 2 is chosen. That choice is interesting because it leads to a data structure that is not predefined in BLISS.

The representation just chosen for a decimal digit array can be diagrammed for the VAX as follows:



This diagram differs from those given in *Section 3.2, "Data Segments"*. In *Chapter 3, "BLISS Values and Data Representations"*, the intent was to represent data structures in a machine-independent way. Here, the intent is to represent the specific layout of the data structure in VAX storage.

The diagram depicts a sequence of bytes in VAX storage. The first line of the diagram (X[1] and X[0]) is the first byte allocated for the array. The second line (. . . and X[2]) is the second byte. The third line suggests successive bytes.

The diagram represents a specific instance of a decimal digit array. The name of the array is X; that is, the value of X is the address of the first byte of the array. The name X is written to the right of the

diagram because of the VAX convention of indexing bits and bytes from right (low order) to left (high order).

The diagram shows that the first element of the vector is called X[0] and contains four bits. That element occupies the four low-order bits of the byte whose address is X. The second element is called X[1] and occupies the four high-order bits of the byte whose address is X. The third element is called X[2] and occupies the four low-order bits of the byte whose address is X+1. The remaining elements of the structure are designated in a similar way.

The name DDA (for decimal digit array) at the top of the diagram refers to the layout of the fields relative to the starting address of the structure. There could be more than one DDA structure in storage at a given time, one at X and others at other addresses.

11.1.3. The Programmed Description of Data Structures

Once the abstract definition and concrete representation of a structure have been designed, the facilities of BLISS can be used to describe and use the structure. The principal facilities are structure-declarations, structure-attributes, and structure-references. However, before these facilities can be described, field-references must be considered.

11.1.3.1. Field-References

A field-reference is a BLISS construct that can designate any portion of storage that is %BPVAL bits or less in size. For example, a field-reference can designate a sequence of 15 bits starting with the second bit of the addressable unit whose address is 3116.

A field-reference has the following form:

```
addr < pos, size, ext >
```

addr

Is interpreted as an addressable-unit address.

pos

Is the number of (least significant) bits skipped before the field begins.

size

Is the number of bits in the field.

ext

Is 0 or 1, depending on whether unsigned or signed extension is used in fetching the contents of the field.

The *ext* parameter can be omitted if unsigned extension is suitable. Sign extension is described in *Section 3.1.3, "Extending Values"*, and a full description of field-references is given in *Section 11.2, "Field-References"*.

Restrictions on the values of *addr*, *pos*, and *size* are different in each BLISS dialect because of differing capabilities of the respective target architectures. Briefly stated, field-references in BLISS–32 can

designate any field of up to %BPVAL bits without regard to address boundaries; while field-references in BLISS–16 and BLISS–36 must designate fields that are completely contained within one fullword.

The BLISS–32 field-references for the decimal digit array X (diagrammed in *Section 11.1.2, "The Concrete Representation of Data Structures"*) are as follows:

```
X<0,4>      !first element, X[0]
X<4,4>      !second element, X[1]
X<8,4>      !third element, X[2]
...         ...
```

The field-reference for the third element is typical; it is interpreted as follows:

```
Find the addressable unit (VAX byte) whose address is X. Start at the low-
order bit of that unit of storage and skip forward across eight bits. Use
 the
next four bits as the field.
```

In this definition, "skip forward" means proceed toward higher order bits and toward higher storage addresses.

Field-references can handle any memory access required in BLISS. However, they are dependent on the concrete representation of data structures. The features described in the following sections are designed to confine the use of field-references to a special place, the structure-declaration, and thus localize the dependence of a program on representation.

11.1.3.2. Structure-Declarations

The following program fragment contains the structure-declaration for BLISS–32 decimal digit arrays (DDAs).

```
STRUCTURE
    DDA[I; N] =
        [(N+1)/2]
        DDA<4*I,4>;
...
OWN
    X: DDA[10];
...
X[5] = .X[6];
```

The first four lines of the example are the structure-declaration. Each line has a different purpose, as follows:

- "STRUCTURE" is the keyword for the declaration.
- "DDA[I; N] =" gives the *structure-name*, DDA, and the formal names I and N. The name I before the semicolon is an *access-formal*, and is used when an instance of the structure is referenced. The name N after the semicolon is an *allocation-formal*, and is used when an instance of the structure is allocated.
- "(N+1)/2" is the *structure-size* and determines the number of addressable units (bytes in this case) allocated for each instance of the structure.
- "DDA<4*I,4>" is the *structure-body* and provides a field-reference for each reference to the structure in the program. Note that, because of dialect-specific differences in field-reference

limitations noted above, this particular structure-body definition is valid in the general case only in BLISS–32.

Observe that in the structure-size and structure-body a fetch operator (.) is not used before a formal name to refer to the value of an actual parameter. In this sense structure formal names are like macro formal names (see *Chapter 16, "Macros"*) and unlike routine formal names (see *Chapter 12, "Routines"*).

11.1.3.3. Structure Allocation

A structure-declaration does not allocate any particular instance of a data structure; it just associates a name with a description of a structure.

An instance of a given structure is allocated when its name is used in a structure-attribute in the declaration of a data segment name. The following declaration allocates a 10-element instance, named X, of a decimal digit array:

```
OWN
    X: DDA[10];
```

The compiler determines how much storage to allocate for X by making a copy of the structure-size, "(N +1)/2", replacing N, the allocation-formal, by 10, and evaluating the expression. The result is 5, and thus five bytes are allocated.

The example structure-size expression is also valid for BLISS–16 (assuming an identical concrete representation for DDA), because the addressable-unit size is the same. The structure-size expression required for BLISS–36, assuming a similar concrete representation for DDA, is given in *Section 11.1.3.7, "Decimal Digit Arrays in BLISS–16 and BLISS–36"*.

11.1.3.4. Structure-References

The following assignment contains two examples of references to the decimal digit array named X:

```
X[5] = .X[6];
```

When the program is compiled, the first structure-reference is replaced by a copy of the structure-body from the declaration of DDA. Then, within the structure-body, DDA is replaced by X and I is replaced by 5. The second structure-reference is compiled in the same way, except that I is replaced by 6. The result is as follows:

```
X<4*5,4> = .X<4*6,4>;
```

The actual-parameter of a structure-reference need not be a numeric-literal as in this example; it can be any expression. For example:

```
X[.J3] = .X[.J3+1];
```

This assignment is expanded by the compiler into the following:

```
X<4*(.J3),4> = .X<4*(.J3+1),4>;
```

In this case, the fields selected depend on the contents of J3 each time the assignment is executed.

Similar examples of the structure-body expression for BLISS–16 and BLISS–36, assuming an identical or similar concrete representation for DDA, are given in *Section 11.1.3.7, "Decimal Digit Arrays in BLISS–16 and BLISS–36"*.

11.1.3.5. REF Structures

It is sometimes useful to manipulate the addresses of data structures. To do this, the compiler needs information about the structures to which the addresses refer. This information is supplied with the REF keyword and an appropriate structure-attribute in the declaration of storage for a structure address. For example:

```
STRUCTURE
    DDA[I; N] =
        [(N+1)/2]
        DDA<4*I,4>;
...
OWN
    X: DDA[10],
    Y: DDA[10];
OWN
    ALPHA,
    PDDA: REF DDA[10];
...
IF .ALPHA EQL 0 THEN PDDA=X ELSE PDDA=Y;
PDDA[5] = .PDDA[6];
```

The interpretation of the final assignment depends on the value of PDDA, and the value of PDDA is determined, at run time, by the contents of ALPHA. If ALPHA contains zero, the assignment is equivalent to the following:

```
X[5] = .X[6];
```

Otherwise it is equivalent to the following:

```
Y[5] = .Y[6];
```

A name that is declared with REF designates a data segment that contains the address of a structure. Because an address always occupies a fullword, a fullword is always allocated for such a name. In the example above, PDDA is the address of a fullword that contains either the address X or the address Y.

When a name that is declared REF is used in a structure-reference (and is therefore followed by a list of parameters in brackets), an extra level of indirection is automatically supplied. For example:

```
PDDA[5] = .PDDA[6];
```

With this assignment, the address of the structure to which a value is assigned is not PDDA but is rather the contents of PDDA. Similarly, the address of the structure from which a value is fetched is not PDDA but is rather the contents of PDDA.

When a name that is declared REF is not used in a structure reference, it is interpreted without the extra level of indirection. If this were not the case, then the contents of a data segment used as a pointer to a structure could not be changed. For example:

```
PDDA = X;
```

With this assignment, the address of the data segment to which a value is assigned is PDDA.

11.1.3.6. Interchangeable Structure-Declarations

You can use different structure-declarations for the same abstract structure at different stages in the development of a program. Three possible declarations for decimal digit arrays are as follows:

- The declaration already considered in the preceding sections is as follows:

```
STRUCTURE
    DDA[I; N] =
        [(N+1)/2]
        DDA<4*I,4>;
```

  This declaration was presented as the one that implements the chosen concrete representation for decimal digit arrays.

- A second declaration of DDA is as follows:

```
STRUCTURE
    DDA[I; N] =
        [N]
        DDA<8*I,8>;
```

  This declaration provides for faster access to the elements but uses twice as much storage.

- A third declaration of DDA follows:

```
STRUCTURE
    DDA[I; N] =
        [N]
        BEGIN
        IF I LSS 0 OR I GTR N-1 THEN ERROR(DDA, I);
        DDA
        END<8*I,8>;
```

  This declaration is oriented toward debugging. Specifically:

  - It uses a full byte (instead of four bits) for each element of the array. Thus the examination of memory is easier.

  - It includes a check on the value of the subscript I to make sure that it is in the range from 0 to N–1. Thus this class of errors is detected automatically.

  This declaration can be used during the development of a program, and one of the previous declarations of DDA can be used for the production version of the same program.

The debugging declaration illustrates an interesting feature of structures. Suppose the following program fragment lies within the scope of the debugging declaration:

```
OWN
    X: DDA[10],
    Y: DDA[20];
...
X[.J] = .Y[.K];
```

The compiler expands the assignment on the last line into the following assignment:

```
BEGIN
IF .J LSS 0 OR .J GTR 9 THEN ERROR(DDA, .J);
DDA
END<8*.J,8>
=
BEGIN
IF .K LSS 0 OR .K GTR 19 THEN ERROR(DDA, .K);
```

```
DDA
END<8*.K,8>;
```

This example shows that the compiler saves the value of the allocation-parameter, N, each time the structure is allocated. For X this value is 10, for Y it is 20. Thus this value can be used in the structure-body and, eventually, in each structure-reference.

11.1.3.7. Decimal Digit Arrays in BLISS–16 and BLISS–36

For a packed four bits per digit decimal digit array in BLISS–36, a different structure-size definition is required for the following reasons:

- The smallest (and only) addressable unit in BLISS–36 is the fullword, rather than the byte as in BLISS–16 and BLISS–32.
- The 36-bit fullword of BLISS–36 can accommodate exactly nine 4-bit digits.

Instead of the BLISS–16/32 structure-size expression "(N+1)/2", which allocates one 8-bit addressable unit for each two elements required plus one unit for an odd final element, the following expression is appropriate for BLISS–36:

```
(N+8)/9
```

This structure-size expression allocates one 36-bit word for each nine elements required plus one word for a final (or only) group of less than nine.

As noted above, the BLISS–32 structure-size expression is also valid for BLISS– 16, since the respective target systems have the same basic storage allocation unit (that is, the byte).

The structure-body definition given for DDA in BLISS–32 needs to be modified in both BLISS–16 and BLISS–36 because neither of these dialects allows the position value of a field-reference to exceed %BPVAL (as it can in BLISS–32). In BLISS–16 the DDA structure-body can be defined as follows:

```
(DDA+I/2)<(I MOD 2)*4,4>
```

Alternatives to this expression, which are logically equivalent but better in terms of object code efficiency, are the following:

1. `(DDA+I/2)<IF I THEN 4 ELSE 0,4>`
2. `(DDA+I/2)<(I AND 1)*4,4>`
3. `(DDA+I/2) <(I^2) AND 4,4>`

These alternatives are listed in order of increasing space efficiency, although the first alternative results in the fastest code sequence.

In BLISS–36 the DDA structure-body can be defined as follows:

```
(DDA+I/9)<(I MOD 9)*4,4>
```

To summarize, the BLISS–16 and BLISS–36 forms of the DDA structure-declaration are the following:

- For BLISS–16 –

  ```
  STRUCTURE
  ```

```
    DDA[I; N] =
        [(N+1)/2]
        (DDA+I/2)<(I^2) AND 4,4>;
```

- For BLISS–36 –

```
STRUCTURE
    DDA[I; N] =
        [(N+8)/9]
        (DDA+I/9)<(I MOD 9)*4,4>
```

The user manual for each BLISS dialect describes, under "Transportability Guidelines", the development of generalized, fully transportable structure-declarations. In particular, it describes a general packed-vector data structure called GEN_VECTOR which produces the same concrete representation described here as DDA on any target system.

11.1.4. Conclusion

All high-level languages provide you with a set of predefined data structures. Some programming languages provide facilities for the definition of new abstract data structures based on predefined data structures. BLISS goes beyond such facilities and provides for the definition of new concrete data structures.

Thus, when the need arises, you can access storage just as freely as an assembly language programmer can. You can designate any addresses, any fields, any bits in storage.

The structure-declaration is the interface between the implementation of a given data structure and its use in the program. On one side of the interface lies the specific layout of the structure, with machine-specific details and an appropriate concern for efficiency. On the other side of the interface are the many references to the structure, each treating it as an abstract, machine-independent entity. For each data structure, communication between the two sides is by a single name, such as DDA used for the example in this section.

Because the predefined structures of BLISS use the same facilities of BLISS as user-defined structures, they provide a point of departure for data description rather than presenting a restrictive barrier.

The BLISS facilities for data structures are unusual and relatively complicated. They depend on the combination of the various declarations, attributes, and references described in this chapter. *Section 11.10, "Predeclared Structures"* and *Section 11.11, "Other Structures"* show how these facilities are combined to define and use specific structures.

11.2. Field-References

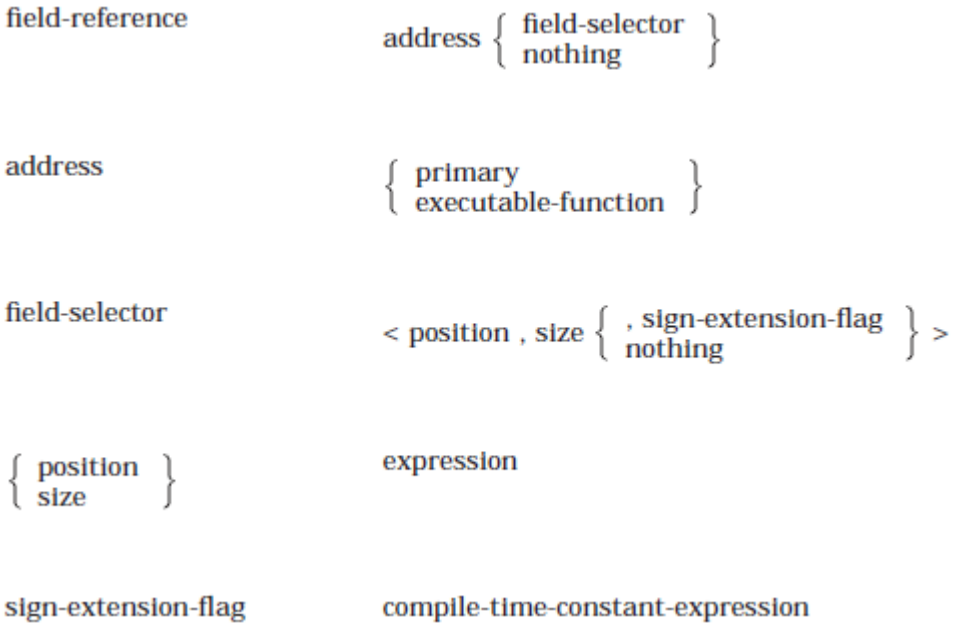
A field-reference designates a sequence of up to %BPVAL bits of storage. It is normally used as the operand of a fetch operator or the left operand of an assignment operator. With certain restrictions, however, a field-reference can be used in any context that requires an address value.

Structure-declarations use field-references to map abstract, machine-independent structures into concrete, machine-specific storage units. Thus, when suitably parameterized, they support the writing of programs that are efficient and yet transportable from one target system to another.

Field-references should be used only in structure-declarations. The use of field-references in any other context introduces machine dependence in a confusing and disorganized way.

Examples of field-references are given in *Section 11.1.3.1, "Field-References"*.

11.2.1. Syntax



In addition to the syntactic rules just given, the following syntactic rules are required:

1. A field-selector is associated with the closest fetch expression.
2. A field-selector that could be part of either an assignment expression or a fetch expression is part of the fetch expression.

An example of an expression to which rule 1 applies is as follows:

```
..BETA<8,8>
```

The expression is correctly interpreted as follows:

```
.(.BETA<8,8>)
```

The following is an *incorrect* interpretation:

```
.(.BETA)<8,8>
```

In this example, the given expression is composed of one fetch expression within another, and rule 1 is needed because one of the fetch expressions does not have a field-selector. In the first interpretation, the field-selector is part of the inner fetch expression, and is, therefore, applied to the data segment whose address is BETA. In the second (nondefault) interpretation, the field-selector is part of the outer fetch expression and, therefore, is applied to the data segment whose address is .BETA.

An example of an expression to which rule 2 applies is as follows:

```
.Q<0,8> = .A+1
```

The expression is correctly interpreted as follows:

```
(.Q<0,8>) = .A+1
```

The following is an *incorrect* interpretation:

```
(.Q)<0,8> = .A+1
```

In the first interpretation, the field-selector is part of the fetch expression and the assignment is made, by default, to a fullword. In the second (nondefault) interpretation, the field-selector is part of the assignment expression, and the fetch is made, by default, from a fullword.

11.2.2. Restrictions

The restrictions on the address, position, and size expression values in a field-selector are different for each BLISS dialect, as follows:

BLISS–16 ONLY

The size of a field may range from 0 to 16 bits, inclusive, but a field must not cross a machine-word boundary. This implies two sets of specific restrictions on the position (p) and size (s) values, as follows:

- If the field-selector is applied to an even-numbered byte address (that is, word-aligned), then the following are true:

  ```
  0 ≤ p
  0 ≤ s ≤ 16
  0 ≤ p+s ≤ 16
  ```

- If the field-selector is applied to an odd-numbered byte address, then the following are true:

  ```
  0 ≤ p
  0 ≤ s ≤ 8
  0 ≤ p+s ≤ 8
  ```

BLISS–32 ONLY

The value of the size expression can range from 0 to 32, inclusive, and the field so specified can cross a longword boundary. More specifically, there is no restriction on the position expression relative to storage-address boundaries, and the restriction on size (s) is as follows:

```
0 ≤ s ≤ 32
```

BLISS–36 ONLY

The value of the size expression can range from 0 to 36, inclusive, but the field so specified must not cross a machine-word boundary. More specifically, the restrictions on position (p) and size (s) are as follows:

```
0 ≤ p
0 ≤ s ≤ 36
0 ≤ p+s ≤ 36
```

The value of the sign-extension-flag must be 0 or 1.

A field-selector must not be immediately followed by another field-selector. For example:

```
.Z<0,16><8,2> = .BETA
```

This is not valid. Parentheses can be used to avoid this restriction. For example:

```
(.Z<0,16>)<8,2> = .BETA
```

This is a valid expression.

Normally a field-reference is the operand of a fetch operator or the left operand of an assignment operator. When a field-reference is used in any other way, it must specify a field that begins on an addressable-unit boundary, which is a position that satisfies the following conditions:

- The value of the position expression must be 0 or 8 in BLISS–16, must be 0 or a multiple of 8 in BLISS–32, and must be 0 in BLISS–36.
- The address expression must not be a register-name.
- The position and size expressions must be compile-time constant expressions.

When the address in a field-reference is a register-name, the field-reference must specify a field that lies entirely within the designated register; that is, the position expression must be greater than or equal to 0 and the sum of the position and size expressions must be less than or equal to %BPVAL.

11.2.3. Default

The default value for the sign-extension-flag is 0.

11.2.4. Semantics

A field-reference specifies a field of up to a fullword (%BPVAL bits) in size relative to a given storage address. Certain aspects of the field-selector semantics are dialect dependent.

In BLISS–16, the field is specified relative to a byte address, and the field must be completely contained in the machine word containing the given byte.

In BLISS–32, the field is specified relative to a byte address, and the field can occur anywhere in storage relative to the given byte.

In BLISS–36, the field is specified relative to a word address, and the field must be completely contained in the given machine word. Depending on the context in which it appears, a field-reference has one of the interpretations given below. These rules do not apply to field-references in the structure-body of a structure-declaration, because the structure-body is not interpreted as part of the declaration of a structure; rather, these rules apply when the structure-body is used in the interpretation of a structure-reference, as described in *Section 11.7, "Ordinary-Structure-References"*, *Section 11.8, "Default-Structure-References"*, and *Section 11.9, "General-Structure-References"*.

- *Fetch context*. If the field-reference is the operand of a fetch expression (defined in *Section 5.1, "Operator-Expressions"*), having the following form:

  ```
  .e2 field-selector
  ```

  then evaluate the fetch expression as follows:

  1. Interpret the address expression, e2, as follows:

     a. If the address is a register-name, then call the register the *selected unit*.

b. Otherwise, let *a* be the value of the address expression. Locate the addressable-unit in storage whose address is *a*. Call this addressable-unit the *selected unit*.

2. Let *p* be the value of the position expression. Locate the sequence of *p* bits that starts with the low-order bit of the selected unit. Call these bits the *offset field*.

3. Let *s* be the value of the size expression. Locate the sequence of *s* bits that immediately follows the offset field. Call these bits the *selected field*.

4. Obtain a fullword value as follows:

   a. If s = %BPVAL, fetch the contents of the selected field.

   b. If 0 < s < %BPVAL, fetch the contents of the selected field and extend it to a fullword as follows:

      i. If the value of the sign-extension-flag is 0, then extend the selected field by adding zero-bits at the left.

      ii. Otherwise, extend the selected field by adding copies of the sign bit (leftmost bit) of the selected field at the left.

   c. If s = 0, use the fullword representation of zero.

5. Use the value just obtained as the value of the fetch expression.

- *Assignment context*. If the field-reference is the left operand of an assignment expression (defined in *Section 5.1, "Operator-Expressions"*), having the following form:

  ```
  e1 field-selector = e2
  ```

  then evaluate the assignment expression as follows:

  1. Locate the selected field of storage, relative to e1, as in steps 1 through 3 for the fetch context.

  2. Let *s* be the value of the size expression and let *v2* be the value of the right operand, e2, of the assignment expression. Store a value as follows:

     a. If *s* = %BPVAL, store *v2* in the selected field.

     b. If 0 < *s* < %BPVAL, store the rightmost *s* bits of *v2* in the selected field.

     c. If s = 0, do not store a value.

  3. Use the fullword value of e2 as the value of the assignment expression.

- *Other contexts*. If a field-reference appears in some other context, then evaluate the field-reference as follows:

  1. Let *a* be the value of the address expression and let *p* be the value of the position. Compute the following:

     ```
     a + p/%BPUNIT
     ```

     Observe that a restriction in *Section 11.2.2, "Restrictions"* requires that the address must not be a register-name, and the value of *p* must be zero or, in the case of BLISS–16/32, a multiple of 8,

so that the value of p/%BPUNIT is an integer. Also observe that the values of the size and sign-extension-flag expressions are not used, but the restrictions on these values still apply.

2. Use the value just computed as the value of the field-reference.

The following considerations apply to the interpretation of field-references:

- The order in which the address, position, size, and sign-extension-flag expressions are evaluated is not defined (see *Section 5.1.4, "Semantics"*).
- The sign-extension-flag is ignored in all contexts except a fetch expression.
- The description of the field-reference just given uses phrases like "sequence of *p* bits that starts with . . . " and "sequence *s* of bits that immediately follows . . . ". Thus it assumes an ordering of bits in storage. That ordering, based on numeric significance, is as follows:

Ordering for BLISS–16 and BLISS–32	
Bit 0	The low-order bit of byte n
⋮	⋮
Bit 7	The high-order bit of byte n
Bit 8	The low-order bit of byte n+1
⋮	⋮
Bit 15	The high-order bit of byte n+1

Ordering for BLISS–32 Only	
Bit 16	The low-order bit of byte n+2
⋮	⋮
Bit 23	The high-order bit of byte n+2
Bit 24	The low-order bit of byte n+3
⋮	⋮

Ordering for BLISS–36	
Bit 0	The low-order bit of word n
⋮	⋮
Bit 35	The high-order bit of word n

- Observe that in BLISS–32, although the selected field cannot be longer than 32 bits, it can occur anywhere in storage, crossing boundaries between bytes, words, or longwords.

11.2.5. Discussion

The BLISS bit numbering convention, defined above, is consistent across the BLISS dialects: bit-position 0 is always the "rightmost" or least significant bit of the specified addressable unit, for all target systems.

Several aspects of field-references are discussed in the following subsections. First, some examples are given to illustrate various cases. Second, the placement of a field-selector in the definition of a structure

is discussed. Finally, the general and fundamental relationship of field-references to expressions is discussed.

11.2.5.1. Examples

Field-references used in fetch and assignment contexts are illustrated throughout this chapter and do not require further elaboration here. However, field-references used in other contexts involve some special considerations.

As stated in *Section 11.2.4, "Semantics"*, a field-reference that is not in a fetch or assignment context computes a value according to the following formula:

```
b + p/%BPUNIT
```

In BLISS–32 and to a limited extent in BLISS–16, such field-references allow you to compute the address at which a field begins. Such address values might be assigned to another data segment for later use or passed as actual-parameters of a routine-call. Observe that the restrictions in such cases (the byte-address is not a register name, position and size are compile-time constant values, and the position is zero or a multiple of 8) assure that the compiler can verify that the field does begin at a byte address and hence, that the above formula can be computed.

Consider the following examples:

Example	**Comment**
A = X	The address of the data segment X is assigned to A.
A = X<0,8>	The address of the data segment X is assigned to A (as in the previous example).
A = X<10,12>	*Invalid*. The field-reference does not designate a field that begins at a byte address.
A = X<8,8>	*Invalid in BLISS–36*; *valid in BLISS–16/32*. The address of the data segment X plus 1 is assigned to A. This field-reference is equivalent to the field-reference (X+1)<0,8>.
A = X<.Y,1>	*Invalid*. The position expression is not a compile-time constant value and, therefore, the field might not begin at a byte address.

Observe that in BLISS–16 the effective range of p/8 is 0 or 1; in BLISS–32, the range of p/8 is unrestricted; and, in BLISS–36, the range of p/36 is always 0. Consequently, the value of a field-reference in BLISS–36 is effectively the same as the address part of the field-reference and the term "p/%BPUNIT" in the formula for the value has no practical utility.

11.2.5.2. Field-References in Structure-Declarations

The definition of a structure-name can include a field-reference as the structure-body (see *Section 11.3, "Structure-Declarations"*), but when the structure-body involves a block, a common error is to place the field-selector inside the block instead of following the block.

An example of correct placement of the field-selector following the block was given in *Section 11.1.3.6, "Interchangeable Structure-Declarations"*; it is repeated here:

```
STRUCTURE
```

```
    DDA[I;N] =
        [N]
        BEGIN
        IF I LSS 0 OR I GTR N-1 THEN ERROR(DDA, I);
        DDA
        END<8*I,8>;
```

Suppose the last two lines of this example are written as follows:

```
...
DDA<8*I,8>
END;
```

This code has a quite different meaning from the one intended. Because the field-reference is contained inside the block, the rule for a field-reference in a context other than a fetch or assignment context always applies. When the structure-reference is used in a fetch or assignment, a fullword fetch or assignment results according to the rules in *Section 5.1, "Operator-Expressions"* (assuming that the restrictions on field-references do not result in an error).

As can be seen in this example, the placement of the field-selector following the block is essential for the desired meaning.

11.2.5.3. Field-References and Expressions in General

Consider again the first two examples in *Section 11.2.5.1, "Examples"* as follows:

```
A = X

A = X<0,8>
```

In both cases, the address of the data segment X is assigned to A. These examples describe a BLISS language design principle that ties field-references and expressions together.

The BLISS rules regarding expressions and data segments given elsewhere in this manual can be restated (in part) in the following way:

1. The declaration of a data segment name associates an implicit, default field-selector with the name, which is determined as follows:

    a. If the data segment is a scalar, then the default field-selector is <0, size, sign> where:

        i. The size value is, in BLISS–16 and BLISS–32, a multiple of %BPUNIT determined by the explicit or default allocation-unit, and in BLISS–36 is simply %BPUNIT, that is, 36.

        ii. The sign value is, in BLISS–16 and BLISS–32, 0 or 1 according to the explicit or default extension-attribute, and in BLISS–36 is always 0.

    b. If the data segment is structured, then the default field-selector is <0, %BPVAL, 0>. This default applies only when the data segment name does not appear in a structure-reference.

2. For any expression other than a data segment name, the default field-selector is <0, %BPVAL, 0>. This default applies only when the expression does not appear as the address-expression of a default-structure-reference.

According to these rules, every expression in a BLISS program can be thought of as having a default field-selector.

When the semantics for field-references given in *Section 11.2.4, "Semantics"* is applied to expressions with default field-selectors as described here, the resulting interpretation is equivalent to the semantics given in *Chapter 5, "Computational Expressions"*. The description given there is used because it is simpler and more intuitive for the common cases. The description given here presents an important part of the conceptual foundation of BLISS.

11.2.5.4. Operations on Scalar Field Values

When all values involved in a calculation occupy fullwords, the programming involved is relatively straightforward. Fullwords accommodate maximum-size BLISS values, and assignment from one fullword to another never modifies a value.

When a scalar field value – a value smaller than a fullword and not part of a data structure – is involved in a calculation, however, problems can arise. They can arise either through assignment of a large value to the small field, or through incorrect extension of the contents of the field. An example of the former type of problem is the inadvertent assignment of a fullword value to a field that is not large enough to accommodate the significant portion of the fullword. Obviously some significance will be lost in the stored result.

The latter type of problem can be more subtle; for example:

```
OWN
    X;
    Y;
...
X<0,8> = -1;
Y = .X<0,8> + 1;
...
```

For purposes of discussion, assume that there is some good reason for using an 8-bit field relative to address X. Because this field occupies less than a fullword, when fetched it is extended before being incremented and assigned to Y. And because the extension for the field is unsigned by default, the extended field value becomes 255 rather than –1. Thus the value of Y becomes 256 rather than 0, presumably not the intended result.

The program fragment does not violate any rules of BLISS; it is valid. However, because it assigns a negative number, –1, to a field that is by implication unsigned, the program fragment is at least ambiguous in its intent, if not incorrect.

Depending on whether the result obtained was or was not the one intended, the program fragment can be altered in one of the following ways:

- Change the numeric-literal from –1 to 255. This change does not affect the value assigned to Y, but does make clear that the result is the expected one.
- Replace the field-selectors shown with <0,8,1>, indicating signed value extension. This change causes 0 to be assigned to Y.

In BLISS–16 or BLISS–32, the problems just described can also arise through the use of an allocation-unit that causes field allocation of a scalar data segment; that is, through the use of BYTE in BLISS–16, or BYTE or WORD in BLISS–32, as an attribute in a data declaration. This is due to the implicit relationship between allocation-units and field-selectors. An equivalent program fragment that uses the BYTE allocation-unit rather than explicit field-references to produce results identical to those described above is given in *Section 5.1.5.3, "Operations on Field Values in BLISS–16/32"*.

11.3. Structure-Declarations

A structure-declaration describes the organization of a data structure. It specifies (or implies) a field-reference for every possible reference to the structure and thus defines the layout of the structure in storage. It also specifies an expression to be used to determine the amount of storage to be allocated when a structure is associated with a name in a data-declaration.

An example of a structure-declaration in each of the BLISS dialects is as follows:

- In BLISS–16 –

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

- In BLISS–32 –

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

- In BLISS–36 –

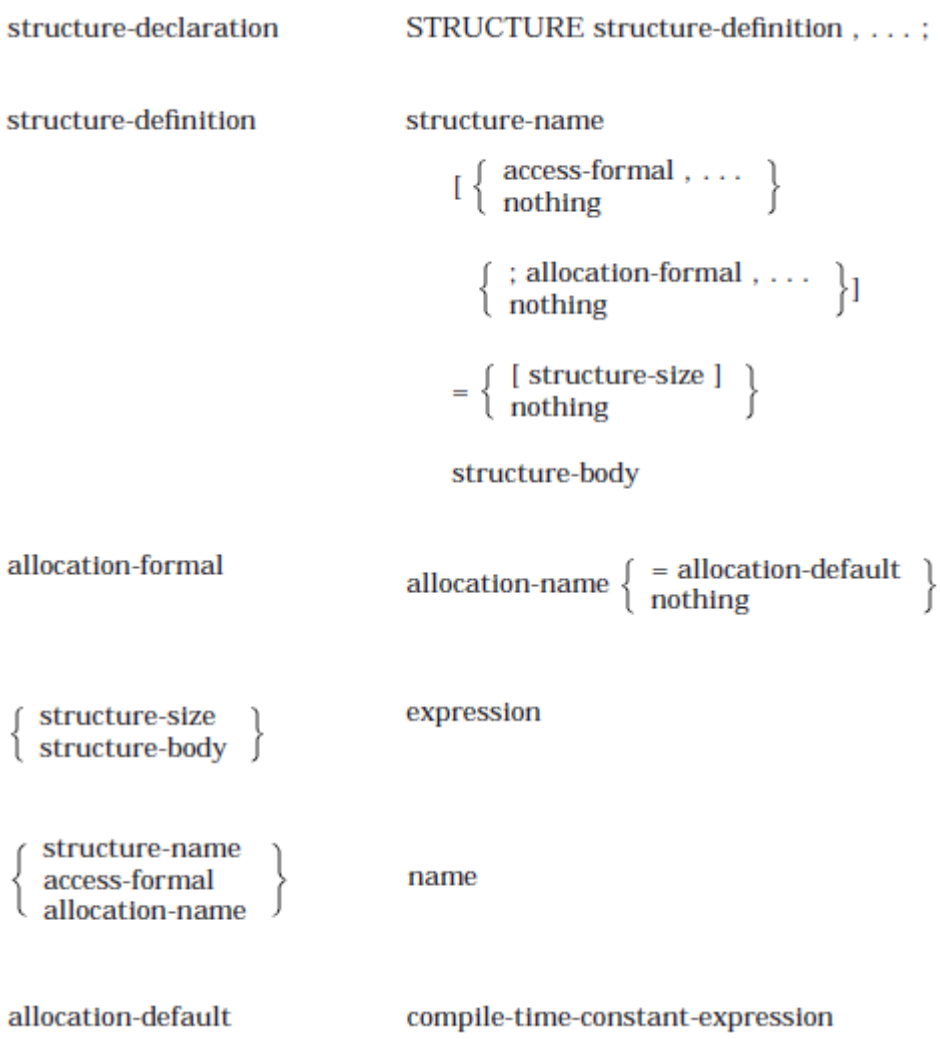
```
STRUCTURE
    VECTOR[I; N] =
        [N]
        (VECTOR+I)<0,36>;
```

These are equivalent declarations of the BLISS predeclared structure named VECTOR, but they do not differ in any significant way from user-written structure declarations.

The access-formal in this declaration is I and the allocation-formals are N and, in BLISS–16/32, UNIT and EXT. UNIT and EXT have default values of %UPVAL and 0, respectively. If in BLISS–16 or BLISS–32 a VECTOR structure-attribute does not specify allocation-actuals for UNIT and EXT, then these default values are used. The structure-size expression is N*UNIT and the structure-body is (VECTOR + I*UNIT)<0,%BPUNIT*UNIT,EXT>.

Observe that in the BLISS–36 VECTOR declaration, the allocation-formals UNIT and EXT are not included. This is because BLISS–36 does not have the corresponding allocation-unit and extension-attribute (used in data-declarations in the other two dialects), and therefore these formal parameters are of no practical use. However, if these formal parameters were expressed in the BLISS– 36 declaration and given their default values of %UPVAL (1 in BLISS–36) and 0 (unsigned-extension), respectively, the BLISS–36 declaration would be not only explicitly equivalent – varying only in the dialect-specific values of %UPVAL and %BPUNIT – but also operationally valid.

11.3.1. Syntax



11.3.2. Restrictions

A primary of a structure-size expression must be either an allocation-name or a compile-time constant expression. When a compile-time constant expression is substituted for each allocation-name in the expression, the resulting expression must be a compile-time constant expression. If the structure-body expression contains a block, only the following declarations can appear in the block:

LOCAL	EXTERNAL LITERAL
STACKLOCAL	EXTERNAL ROUTINE
REGISTER	LITERAL
EXTERNAL	

11.3.3. Semantics

The structure-size expression of a structure-declaration is used by the compiler when the structure name appears in a structure-attribute of a data-declaration. It specifies the number of addressable units to allocate for the declared data segment.

The structure-body is used each time a structure-reference appears in an expression. It specifies a replacement for the structure-reference that consists of an expression. Observe that a field-reference is one form of expression.

The use of these portions of the structure-definition is described in the following sections on structure-attributes and storage allocation (*Section 11.4, "Structure-Attributes and Storage Allocation"*) and structure-references (*Section 11.7, "Ordinary-Structure-References"*, *Section 11.8, "Default-Structure-References"*, and *Section 11.9, "General-Structure-References"*).

11.4. Structure-Attributes and Storage Allocation

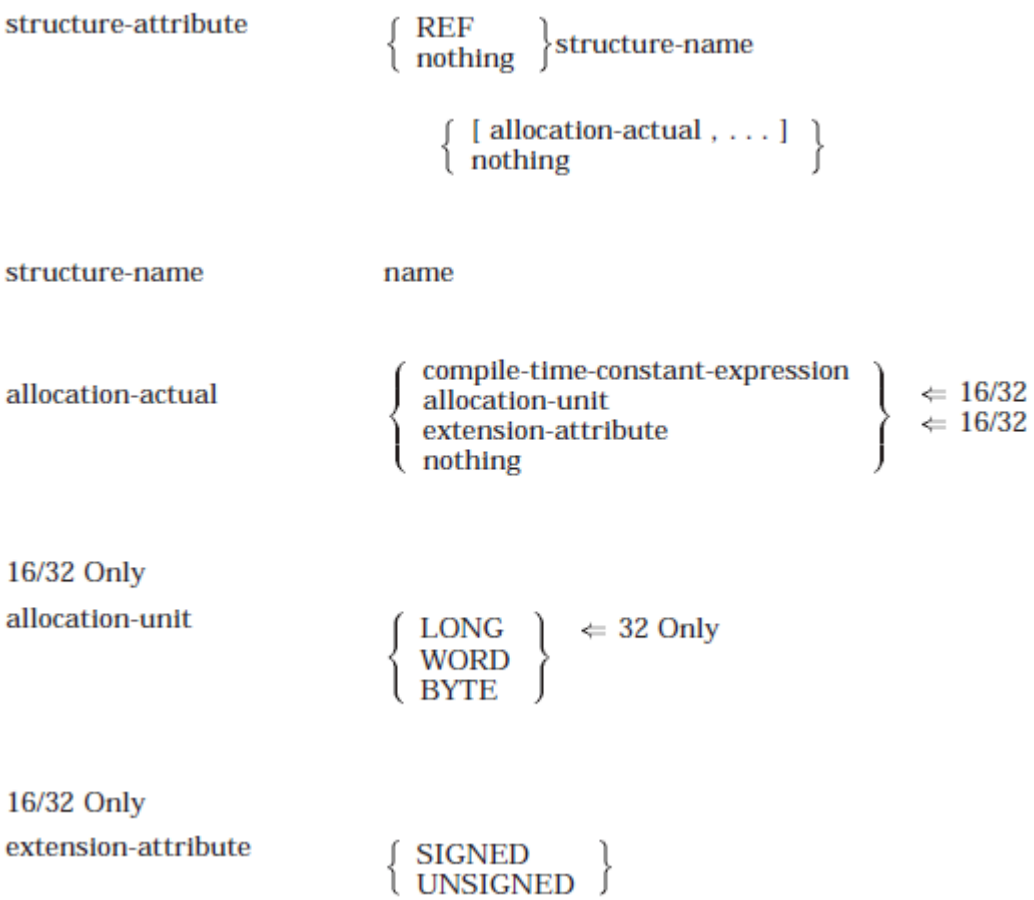
The form of a data segment is determined when its name is declared. If a structure-attribute appears in the declaration, then that structure-attribute determines the structure of the data segment both for purposes of storage allocation and access. If no structure-attribute appears, then the data segment is assumed to be a scalar.

A structure-attribute in the declaration of a name provides two kinds of information. First, it provides a structure-name and thus associates a structure-definition with the name of the data segment. Second, it provides the allocation-actual parameters for the structure-definition, and thus specifies the number of addressable units of storage to be allocated for the data segment.

Observe that the parameters in a structure-attribute are positional; that is, the formal names given in the structure-declaration are not used as keywords in a structure-attribute.

The complete syntax and semantics of the declarations in which a structure-attribute can appear are given in the chapters on data declarations (*Chapter 10, "Data Declarations"*) and on binding (*Chapter 14, "Binding"*). This section describes only the structure-attribute itself and how it is used to determine the size of a structured data segment.

11.4.1. Syntax



11.4.2. Restrictions

BLISS–16/32 ONLY

An allocation-unit used directly as an attribute cannot appear in the same declaration as a structure-attribute. Similarly, an extension-attribute used directly as an attribute cannot appear in the same declaration as a structure-attribute.

Unless the structure-attribute begins with REF or is in an EXTERNAL, MAP, or BIND declaration, the following conditions apply:

- A structure-size expression must appear in the definition of the structure-name.
- A non-null allocation-actual parameter must be given for each allocation-name that appears in the structure-size expression and does not have an allocation-default.

A non-null allocation-actual parameter must be given for each allocation-name that appears in the structure-body and does not have an allocation-default.

11.4.3. Semantics

The allocation of a structure is performed by the compiler as follows:

1. If in BLISS–16 or BLISS–32 an allocation-unit or extension-attribute keyword appears as an allocation-actual, it is replaced by a constant value as follows:

Keyword	Replaced by
LONG	4 – 32 Only
WORD	2
BYTE	1
SIGNED	1
UNSIGNED	0

2. The allocation-actual parameters are evaluated and the values are associated with the corresponding allocation-names in the specified structure-definition.

3. Any allocation-name that does not have a value already associated with it from step 2, but does have an allocation-default value, is associated with its default value.

4. The amount of storage to allocate for the declared name is determined as follows:

   a. If the structure-attribute appears in an EXTERNAL, MAP, or BIND declaration, then no storage is allocated.

   b. If the structure-attribute begins with the keyword REF, then one fullword of storage is allocated.

   c. Otherwise, the structure-size expression is evaluated using the values that are associated with each of the allocation-formal names. The resulting value specifies the number of addressable units of storage that are allocated.

5. The structure-name and the values associated with each allocation-name are recorded with the data-segment name being declared, for use when the data-segment is referenced.

11.5. Field-Declarations

The FIELD declaration is used to define names of fields in BLOCK and BLOCKVECTOR predeclared structures, and in user-defined structures that are similar to BLOCK. A BLISS–36 example of a field-declaration is as follows:

```
FIELD
    DCB_FIELDS =
        SET
        DCB_A = [0,0,36,0],
        DCB_B = [1,0,6,0],
        DCB_C = [1,6,12,0],
        DCB_D = [1,18,18,0],
        DCB_E = [2,0,36,0]
        TES;
```

The field-names declared here are DCB_A, DCB_B, and so on. Each name can be used as a parameter in a structure-reference to represent a sequence of four access-actuals. For example, DCB_A can be used to represent "0,0,36,0". In other examples, the field-names might represent more or less than four access-actuals.

The example field-declaration also provides a field-set-name, DCB_FIELDS. This name is used to refer to the field-names collectively, when, for example, they must be mentioned in a field-attribute.

The field-declaration is a special-purpose facility that can best be explained in the context of a complete example of structure declaration and use. Such an example is given in *Section 11.10.3, "BLOCK Structures"*.

11.5.1. Syntax

```
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-set-name }         name
{ field-name     }

field-component            compile-time-constant-expression
```

11.5.2. Restrictions

A field-name can be used only as an access-actual parameter of a structure-reference, a parameter of a field-attribute, or in the %FIELDEXPAND lexical-function.

A field-set-name can be used only as a parameter of a field-attribute.

A field-name cannot be used for its own definition in a field-component.

11.5.3. Semantics

The field-declaration defines names for use as access-actual parameters of structure-references to designate fixed fields in fixed data structures. As a notational convenience, a set of such field-names can be declared and referred to by a single name. Observe that both field-names and field-set-names follow the normal rules concerning scope and uniqueness of names; there is no concept like the "qualified names" of COBOL or PL/I.

When a field-name appears as an access-actual parameter of a structure-reference, it is replaced by the list of field-component values from the field-definition. See example in *Section 11.10.3.5, "BLOCK Field-Declarations"*. These values provide one or more of the access-actual parameters used in the evaluation of the structure-reference. A field-name need not itself supply all of the actual parameters required for the reference. While this replacement has some of the characteristics of a macro expansion, field-names are not macro-names; in particular, a field-name is not valid in contexts other than a structure-reference.

The field-attribute specifies the set of field-names that can appear in ordinary-structure-references for the indicated data segment. If no field-attribute is given, then no field-name is valid.

Any field-name can be used in a general-structure-reference.

11.6. Field-Attributes

A field-attribute is used in the declaration of a structured data segment name; that is, in the same declaration with a structure-attribute. The field-attribute supplies field-names for some or all of the fields in the structured data segment, either directly by listing field-names or indirectly by giving one or more field-set-names, or both.

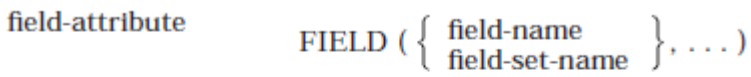
An example of the use of a field-attribute follows:

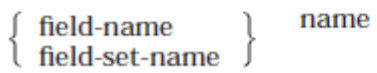
```
OWN
    ALPHA: BLOCK[DCB_SIZE] FIELD(DCB_FIELDS);
```

In this example, the field-attribute associates the field-set-name DCB_FIELDS with the data segment name ALPHA.

Like the field-declaration, the field-attribute can best be explained in the context of a complete example of structure declaration and use. Such an example is given in *Section 11.10.3, "BLOCK Structures"*.

11.6.1. Syntax





11.6.2. Restrictions

Although a field-set-name can appear as a field-attribute parameter in a data segment declaration, it cannot be used in a structure-reference to the data segment. The individual field-names associated with the field-set-name must be used instead.

A field-attribute can be used only in a declaration that also has a structure-attribute.

11.6.3. Semantics

A field-attribute specifies the set of field-names that can appear in an ordinary-structure-reference to the data segment declared with the given field-attribute. A field-set-name in a field-attribute implies a defined set of field-names that can so appear. If no field-attribute is given, then no field-name is valid in such a reference.

11.7. Ordinary-Structure-References

A structure-reference is used to access a part of a structured data segment. The part of the segment that is accessed is determined by the access-actual parameters in the structure-reference. For example, a structure-reference for a vector has one access-actual parameter that specifies the element of the vector to be accessed.

Three kinds of structure-reference are provided: ordinary, default, and general. The ordinary-structure-reference is by far the most commonly used form. It gives the name of a data segment and relies on the

compiler to determine the appropriate structure from the declaration of the segment name. A default-structure-reference is similar, but the address of the data segment is given by an expression, often a preceding ordinary- or default-structure-reference, and relies on the compiler to determine the structure from the default structure specification given in a switches-declaration or module-switch. A general-structure-reference is self-contained. It gives all the information necessary for the access.

Suppose the declaration of A is as follows:

```
OWN A: VECTOR[10];
```

An example of an ordinary-structure-reference follows:

```
A[.J]
```

The compiler uses the declaration of A to find the kind of structure that is being accessed. This ordinary-structure-reference is a reference to a VECTOR that consists of 10 elements. The structure-body that is declared for VECTOR is used in combination with the allocation-actuals in the declaration of A and the access-actuals in the structure-reference to determine the field-reference for the appropriate element of the vector. Suppose the following set of declarations is given:

```
OWN A:  VECTOR[10];
SWITCHES STRUCTURE (BLOCK [1]);
FIELD  FL = [0,0,%BPVAL/2,0],
       FR = [0,%BPVAL/2,%BPVAL/2,0];
```

An example of a default-structure-reference follows:

```
A[.J][FL]
```

The compiler processes the initial ordinary-structure-reference, A[.J]. The field-reference that results is then used as the address part of a subsequent structure-reference. The compiler uses the specification of the default structure in the switches-declaration to find the kind of structure that is being accessed. In this example the default-structure-reference is a block that consists of one fullword. The structure-body that is declared for the block is used in combination with the allocation-actuals in the default structure specification in the SWITCHES declaration to determine the field-reference for the appropriate field in the jth element of segment A.

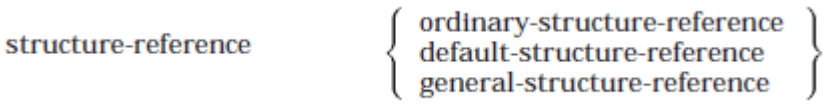
An example of a general-structure-reference follows:

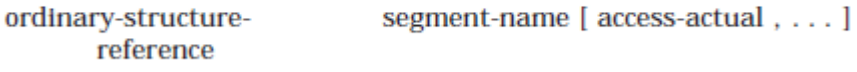
```
VECTOR[A, .J; 10]
```

This general-structure-reference is equivalent to the ordinary-structure-reference given above.

Ordinary-structure-references are described in this section. Default- and general-structure-references are described in the next two sections.

11.7.1. Syntax





segment-name name

access-actual { field-name | expression | nothing }

11.7.2. Restrictions

A structure-attribute must be associated with the segment-name.

If field-names are used as access-actuals in the structure-reference, then a field-attribute designating those field-names must be associated with the segment-name. An access-actual parameter must be given for each access-formal name that appears in the structure-body of the associated structure-definition.

11.7.3. Semantics

An ordinary-structure-reference is interpreted as follows:

1. Use the segment-name to obtain the structure-body of the associated structure-definition and to obtain the values associated with each of the allocation-names for that segment-name.
2. If the structure-attribute for the segment did not include the keyword REF, then determine the value of the data segment name (which is the address of the data segment) and associate that value with the structure name.

   If the structure attribute did include the keyword REF, then fetch the fullword contents of the segment-name and associate that value with the structure name.
3. If one or more access-actuals is a field-name, replace each field-name with its defined sequence of field-component values. This replacement may increase the number of access-actual expressions in the resulting structure-reference.
4. Evaluate the access-actual expressions and associate the ith access-actual value with the ith access-formal name in the structure definition. The order of evaluation of the access-actual expressions is not defined (see *Section 5.1.4, "Semantics"*).
5. Evaluate the structure-body using the values associated with each of the allocation-formal names, the access-formal names, and the structure-name.
6. Use the resulting expression (which is typically a field-reference) in place of the structure-reference.

11.7.4. Discussion

An important characteristic of structure-references is that the access-actual expressions in a structure-reference are each evaluated exactly once. The resulting value is used in the structure-body evaluation in each place that the access-formal appears.

Consider the following declarations:

```
EXTERNAL ROUTINE
    X,
    Y,
    F;
STRUCTURE
```

```
    XYZ[A;B] =
        [B]
        (XYZ+X(A)+Y(A));
OWN ABC: XYZ[4];
```

Given these declarations, the structure-reference ABC[F()] is logically equivalent to the following:

```
BEGIN
LOCAL TEMP;
TEMP = F();
X(.TEMP) + Y(.TEMP)
END
```

The routine F is called once in the structure-reference ABC[F()] and the resulting value is used twice.

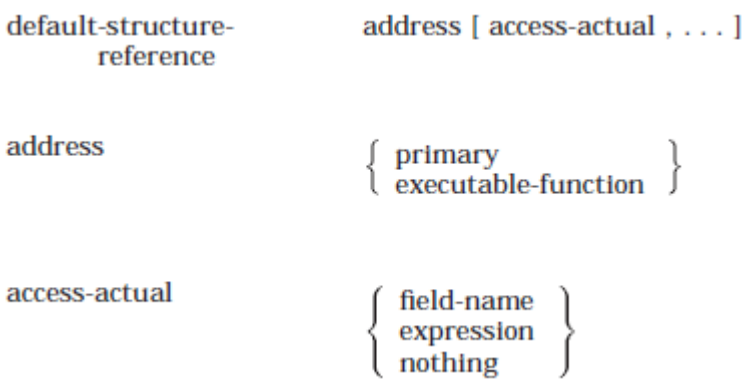
Because structure-references are handled by the compiler in a manner similar to macro expansions and they are, in fact, compiled to inline code, it is natural to think of structure-references as macro calls; however, this example shows that the interpretation of the actual parameters is more similar to that for routine-calls.

11.8. Default-Structure-References

A default-structure-reference is used when an ordinary-structure-reference cannot provide the required field-reference. This usage arises when the address of the accessed data segment is an expression, so that the name of the data (which is part of an ordinary-structure-reference) is not known. When this occurs frequently in a block or module, it can be convenient to give a default structure-attribute in a switches-declaration or module-switch to provide the structure information to be used for all such occurrences.

An example of a default-structure-reference has already been given in the introduction of *Section 11.7, "Ordinary-Structure-References"*. A more extensive example is given in *Section 11.11.7, "General-Purpose Structures for Default Structure References"*.

11.8.1. Syntax



11.8.2. Restrictions

The address of a default-structure-reference must not be the name of a data segment declared with a structure-attribute. If the address is the name of a data segment declared with a structure-attribute, then the structure-reference is an ordinary-structure-reference and is interpreted as described in *Section 11.7, "Ordinary-Structure-References"*.

A default-structure-reference must only occur in the scope of a nonempty STRUCTURE switch-item (see *Section 11.8.2, "Restrictions"*).

An access-actual parameter must be given for each access-formal name that appears in the structure-body of the definition of the default structure.

11.8.3. Semantics

A default-structure-reference is interpreted as follows:

1. Use the default structure-attribute to get the structure-body of the associated structure-definition and to get the allocation-actual values associated with each of the allocation-names of the structure.

2. If the default structure-attribute does not include the keyword REF, then associate the value of the address of the structure reference with the structure-name. If the default structure-attribute does include the keyword REF, then fetch the fullword contents of the address value, and associate the result with the structure-name.

3. If one or more access-actuals is a field-name, replace each field-name with its defined sequence of field-component values. This replacement may increase the number of access-actual expressions in the resulting structure-reference.

4. Evaluate the access-actual expressions and associate the ith access-actual value with the ith access-formal name in the structure-definition. The order of evaluation of the access-actuals is not defined (see *Section 5.1.4, "Semantics"*).

5. Evaluate the structure-body using the values associated with each of the allocation-formal names, the access-formal names, and the structure-name.

6. Use the resulting expression (which is typically a field-reference) in place of the structure-reference.

11.8.4. Discussion

Default-structure-references are very similar to ordinary-structure-references. The differences are as follows:

1. A default-structure-reference uses the structure information established in a default structure-attribute, and hence, must occur in the scope of a nonempty STRUCTURE switch-item. In contrast, an ordinary-structure-reference uses the structure information associated with the declaration of a data segment name and is independent of whether or not a default structure-attribute is established.

2. A default-structure-reference permits any field-name to be used as an access-actual parameter. In this respect it is like a general-structure-reference; see *Section 11.9, "General-Structure-References"*. There is no way to specify a default field-attribute to go with the default structure-attribute. In contrast, an ordinary-structure-reference permits only those field-names that are given in the field-attribute of the data segment declaration.

   Observe that when an ordinary- or default-structure-reference occurs as the address part of another default-structure-reference, the interpretation occurs from left to right. That is, structure-references of the following forms are equivalent:

   ```
   exp[ actuals ,... ]  [ actuals ,... ]

   ( exp [ actuals ,... ] )  [ actuals ,... ]
   ```

   Also observe that such a structure-reference is a primary and is interpreted before any operators are applied. For example, the following are equivalent:

   ```
   X = .Y[1][2]
   ```

```
X = .(Y[1])[2]
```

The following are also equivalent:

```
X = ..Y[1][2][3]

X = ..((Y[1])[2])[3]
```

Consider the following block:

```
BEGIN
SWITCHES STRUCTURE(VECTOR[10]);
OWN X;
...
X[0] = 1;                       !Valid
...
     BEGIN
     SWITCHES STRUCTURE ();
     ...
     X[0] = 1;                  !Invalid
     ...
     END
...
END
```

The declaration of X in this example does not associate the structure-attribute VECTOR[10] with X. Segment X is a scalar by default and is allocated a single fullword.

The first occurrence of X[0], in the fifth line of the example, is a valid default-structure-reference. It cannot be an ordinary-structure-reference because no structure-attribute is associated with X. The second occurrence of X[0], in the tenth line of the example, is invalid because the default structure-attribute is empty and, as before, there is no structure-attribute associated with X. As another example, consider the following block:

```
BEGIN
SWITCHES STRUCTURE(VECTOR[100]);
OWN X:  BITVECTOR[20];
...
X[.I] = 1;
...
(X)[.I] = 1;
END
```

In this example, the structure-reference X[.I] is an ordinary-structure-reference because the structure-attribute BITVECTOR[20] is given in the declaration of X. Thus, the interpretation of the structure-reference uses the BITVECTOR structure (and not the VECTOR structure).

The structure-reference (X)[.I] is a default-structure-reference because (X), the base address of the reference, is not a data segment name. The value of the expression (X) is the same as the value of X, but the BITVECTOR structure-attribute associated with X is lost in the evaluation of the expression (X), just as it is in the evaluation of the expressions (X+4) and (X+0). Thus, the interpretation of the structure-reference (X)[.I] uses the VECTOR structure (and not the BITVECTOR structure).

The above examples illustrate how it is possible to be confused about whether a structure-reference is ordinary or default when the address is a data segment name. For this reason, default-structure-references should be used cautiously and only when there is a very good reason.

A default-structure-reference provides no capability that cannot also be achieved with a general-structure-reference. It is strictly a notational and stylistic convenience.

More examples are given in *Section 11.11.7, "General-Purpose Structures for Default Structure References"*.

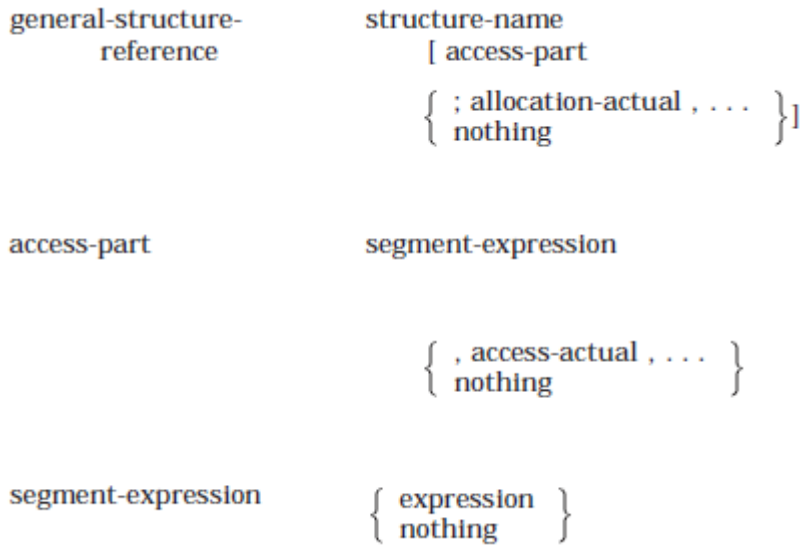
11.9. General-Structure-References

A general-structure-reference is used when an ordinary-structure-reference cannot provide the required field-reference. This usage arises in two ways. First, a general-structure-reference must be used when the address of the accessed data segment is an expression, so that the name of the data segment (which is part of an ordinary-structure-reference) is not known. Second, a general-structure-reference can be used to access a given data segment using a different structure-definition than that which is associated with the name of the data segment. An example of the second use of a general-structure-reference is given in the following block:

```
BEGIN
STRUCTURE
    ARRAY[I, J; M, N] =
        [M*N*%UPVAL]
        (ARRAY+(I*N+J)*%UPVAL);
OWN ALPHA: VECTOR[200];
...
ARRAY[ALPHA,.I,.J;50,4] = 0;
...
END
```

The general-structure-reference interprets the vector ALPHA as a two-dimensional array according to the structure-declaration for ARRAY. The declaration of this two-dimensional array structure is discussed in *Section 11.11.3, "Two-Dimensional Array Structures"*.

11.9.1. Syntax



The syntactic names structure-name, access-actual and allocation-actual are defined in *Section 11.3, "Structure-Declarations"* and *Section 11.4, "Structure-Attributes and Storage Allocation"*.

11.9.2. Restrictions

If the structure-name appears in the structure-body of the definition of the structure-name, then the segment-expression must be nonempty.

An access-actual parameter must be given for each access-formal name that appears in the structure-body of the definition of the structure-name.

An allocation-actual must be given for each allocation-name that appears in the structure-body and that does not have an allocation-default.

11.9.3. Semantics

A general-structure-reference is interpreted as follows:

1. Use the structure-name to get the structure-body for the declaration of that name.

2. If one or more of the access-actuals is a field-name, replace each field-name with its defined sequence of field-component values. This replacement may increase the number of access-actual expressions in the resulting structure-reference.

3. Evaluate the segment-expression and associate the value with the structure-name in the structure definition.

4. Evaluate the access-actual expressions and associate the ith access-actual value with the ith access-formal name in the structure definition.

5. In BLISS–16 or BLISS–32, if an allocation-unit or extension-attribute keyword appears as an allocation-actual, replace it by a constant value as follows:

Keyword	**Replaced by**
LONG	4 – 32 Only
WORD	2
BYTE	1
SIGNED	1
UNSIGNED	0

6. Evaluate the allocation-actual expressions and associate the ith allocation-actual value with the ith allocation-formal name in the structure definition. Observe that each allocation-actual is a compile-time constant value.

7. Any allocation-formal that does not have a value already associated with it from the previous step, but does have an allocation-default value specified, is associated with that default value.

8. Evaluate the structure-body using the values associated with the access-formals, allocation-formals, and the structure-name.

9. Use the resulting expression (which is typically a field-reference) in place of the structure-reference.

The order of evaluation of the segment-expression and access-actual expressions is not defined (see *Section 5.1.4, "Semantics"*).

The interpretation of a general-structure-reference combines the relevant parts of the rules for interpretation of an ordinary-structure-reference and the structure-attribute for a given data segment.

11.9.4. Discussion

A general-structure-reference of the form:

```
structure-name [ segment, access ,... ; allocation ,... ]
```

is equivalent to the following field-reference:

```
BEGIN
BIND base = address
         : structure-name [ allocation ,... ];
base [ access ,... ]
END field-selector
```

base

Is an arbitrary unique name created for the purpose of this discussion.

address

Is the address part of the field-reference in the structure-body of the declaration of the structure-name.

field-selector

Is the field-selector part of the field-reference in the structure-body of the declaration of the structure-name. As the syntax of *Section 11.2, "Field-References"* and *Section 11.3, "Structure-Declarations"* show, a field-selector is optional.

The BIND declaration is described in *Section 14.3, "Bind-Data-Declarations"*.

As with an ordinary-structure-reference, the parameters of a general-structure-reference are evaluated once, and the resulting values can be used more than once (see *Section 11.7.4, "Discussion"*).

Unlike an ordinary-structure-reference, however, any field-name can be used as an access-actual of a general-structure-reference. There is no way to designate a specific set of field-names that are valid; that is, there is nothing analogous to the field-attribute for general-structure-references.

A general-structure-reference does not include (or need) anything analogous to the REF keyword in a structure-attribute. You achieve the same effect by explicitly indicating the extra fetch in the segment-expression. For example:

```
OWN
    A: VECTOR[10],
    B: REF VECTOR INITIAL(A);
...
A[1] = 1;
VECTOR[A,1;10] = 1;
B[1] = 1;
VECTOR[.B,1;10] = 1;
```

All four assignments have the same effect; namely, they assign one to the second element of A. The first two assignments show the corresponding ordinary- and general-structure-references for the non-

REF structure A. The second two assignments show the corresponding ordinary- and general-structure-references for the REF structure B.

11.10. Predeclared Structures

The structures most commonly used in system programming are predeclared in BLISS. The use and interpretation of each of these structures has already been introduced in *Chapter 3, "BLISS Values and Data Representations"* and used in examples. This section presents the definition of each of these structures.

It is possible to write structure-declarations that are equivalent to the four predeclared structures; they are predeclared in BLISS as a convenience and to foster the use of uniform names for these common structures.

The predeclared structures are the following:

Structure-Name	Usage
VECTOR	A vector of signed or unsigned elements of uniform size (bytes or words in BLISS–16; bytes, words, or longwords in BLISS–32; and words in BLISS–36)
BITVECTOR	A vector of one-bit elements
BLOCK	A sequence of varying-sized fields
BLOCKVECTOR	A vector of blocks

The declaration and use of the predeclared BLOCK structure is discussed here in detail because of its fundamental nature (along with VECTOR, discussed previously). The BITVECTOR and BLOCKVECTOR structures are discussed more briefly because they are straightforward variations of the VECTOR and BLOCK structures.

11.10.1. VECTOR Structures

A VECTOR structure is a sequence of elements of the same size. The number of elements (n) is the extent of the vector. The elements are numbered from 0 to n–1. The generalized form of the structure-declaration is as follows:

```
STRUCTURE
    VECTOR[I; N, UNIT=%UPVAL, EXT=0] =
        [N*UNIT]
        (VECTOR+I*UNIT)<0,%BPUNIT*UNIT,EXT>;
```

When this generalized declaration is made dialect specific, the resulting (actual) structure-declaration of VECTOR in each dialect is as follows:

- In BLISS–16 –

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

- In BLISS–32 –

```
STRUCTURE
```

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

- In BLISS–36 –

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

The formal names of the structure-declaration have the following meanings:

Formal Name	Meaning
I	The number of the element to be referenced.
N	The number of elements in the vector.
UNIT	The number of addressable-units in each element. The valid values vary with the target system: 1 or 2 for BLISS–16, and 1 through 4 for BLISS–32. Because the only valid value would be 1 in BLISS–36, the formal name UNIT is omitted in that dialect. The default value, %UPVAL, implies a fullword.
EXT	The sign-extension rule to be used for fetching elements. The valid values are 0 and 1. The default is 0, that is, unsigned. Sign-extension of a fullword is not meaningful; thus, the formal name EXT is omitted in BLISS–36.

Example uses of this structure as structure-attributes in declarations are as follows:

Example	Interpretation
VECTOR[10]	A vector of 10 fullwords
VECTOR[10,WORD]	A vector of 10 unsigned words in BLISS–16/32
VECTOR[20,BYTE,SIGNED]	A vector of 20 signed bytes in BLISS–16/32
REF VECTOR[5]	A reference to a vector of 5 fullwords
VECTOR[20,3]	A vector of twenty 3-byte elements, in BLISS–32 only

11.10.2. BITVECTOR Structures

A BITVECTOR is a sequence of one-bit elements that are densely packed in storage. The number of elements (n) is the extent of the bitvector. The elements are numbered from 0 to n–1. The generalized form of the structure-declaration is as follows:

```
STRUCTURE
    BITVECTOR[I; N] =
        [(N+(%BPUNIT-1))/%BPUNIT]
        (BITVECTOR+I/%BPUNIT)<I MOD %BPUNIT,1,0>;
```

The actual, dialect-specific forms of this structure-declaration are as follows:

- In BLISS–16:

```
STRUCTURE
```

```
    BITVECTOR[I; N] =
        [((N+7)/8)]
        (BITVECTOR+(I^-3))<I AND 7,1,0>;
```

- In BLISS–32 the following variation is used to take advantage of the less restrictive field-references for better code quality:

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

- In BLISS–36:

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

The formal names of this structure have the following meanings:

Formal Name	Meaning
I	The number of the element to be referenced.
N	The number of elements in the vector.

Examples of uses of this structure as structure-attributes in declarations are as follows:

Example	Interpretation
REF BITVECTOR[8]	A reference to a vector of eight 1-bit elements
BITVECTOR[60]	A vector of sixty 1-bit elements

Observe that the second data segment would occupy eight bytes of PDP–11 or VAX storage, and would leave the four high-order bits of the last byte unused. On the DECsystem–10/20 the first data segment would occupy one word with 28 high-order bits unused; the second would occupy two words with 12 high-order bits of the second word unused.

11.10.3. BLOCK Structures

A BLOCK structure is a sequence of components. The individual components of a block can be of various sizes. The generalized form of the structure-declaration is as follows:

```
STRUCTURE
    BLOCK[O, P, S, E; BS, UNIT=%UPVAL] =
        [BS*UNIT]
        (BLOCK+O*UNIT)<P,S,E>;
```

The actual, dialect-specific forms of this structure-declaration are as follows:

- In BLISS–16:

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

- In BLISS–32:

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

- In BLISS–36:

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

The formal names of this structure have the following meanings:

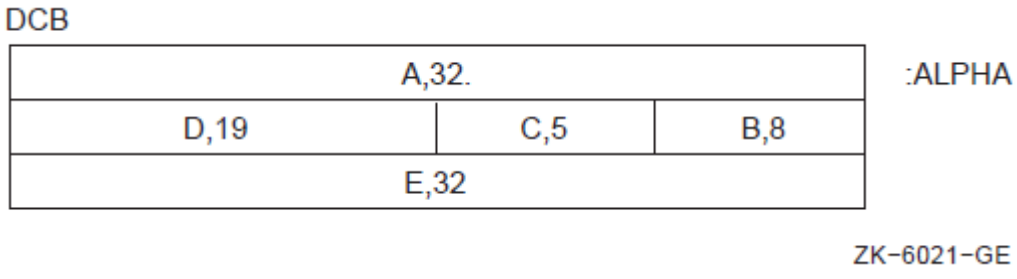
Formal Name	Meaning
O	The offset to the addressable-unit in which the field begins.
P	The bit offset from the addressable-unit to the field beginning.
S	The size of the field in bits. Valid values are 0 to %BPVAL.
E	The extension flag. Valid values are 0 for zero extension and 1 for sign extension.
BS	The number of allocation units needed to represent the block (that is, the block size).
UNIT	The size of the allocation-unit and offset in terms of addressable units. Valid values vary with the target system: 1 or 2 for BLISS–16, 1 through 4 for BLISS–32, and 1 only in BLISS–36 (the formal-name UNIT is omitted in that dialect). The default is %UPVAL, that is, a fullword.

Blocks are conventionally allocated in fullword units for most efficient operation of the hardware. Using default fullword allocation also facilitates transportability of BLISS programs.

11.10.3.1. A Typical Byte-Oriented BLOCK Structure

An example of a typical block on a byte-oriented target system (PDP–11 or VAX) is considered in detail in the following paragraphs. The block is named ALPHA and has five components, named A, B, C, D, and E. The VAX target system and BLISS–32 dialect are assumed for the purposes of this example as they provide the richest basis for explanation of the underlying BLISS structure mechanisms. A BLISS–36 example would be somewhat simpler because addressable byte boundaries are not considered. Analogous code fragments for BLISS–36 are shown in this discussion where appropriate.

The layout of the example block in VAX storage is as follows:



This diagram uses the notation introduced in *Section 11.1.2, "The Concrete Representation of Data Structures"*.

The name DCB refers to the layout of the fields relative to the starting address of the block. Thus there could be more than one DCB block in storage at a given time, one at ALPHA and others at other addresses.

The block is divided into five components, and the name and size are given for each component. Component A contains 32 bits and occupies the four bytes whose addresses are ALPHA through ALPHA+3. Component B contains 8 bits and occupies the byte at ALPHA+4. Component C contains 5 bits and occupies the 5 low-order bits of the byte at ALPHA+5. Component D contains 19 bits and occupies the remaining bits of the byte at ALPHA+5 as well as the next two bytes. Component E occupies the next longword.

11.10.3.2. BLOCK Field-References

Each component of a block has a field-reference. The field-references for DCB are as follows:

Component	Field-Reference	Analogue for BLISS–36
A of ALPHA	(ALPHA+0)<0,32,0>	(ALPHA+0)<0,36,0>
B of ALPHA	(ALPHA+4)<0,8,0>	(ALPHA+1)<0,8,0>
C of ALPHA	(ALPHA+4)<8,5,0>	(ALPHA+1)<8,5,0>
D of ALPHA	(ALPHA+4)<13,19,0>	(ALPHA+1)<13,23,0>
E of ALPHA	(ALPHA+8)<0,32,0>	(ALPHA+2)<0,36,0>

For example, the field-reference expression for component D of ALPHA is interpreted by locating the byte whose address is (ALPHA+4) and then applying the field-selector <13,19,0> at that position in memory. The field-selector starts at the low-order (rightmost) bit of the designated byte, then skips 13 bits (first parameter) to the left, then selects the next 19 bits (second parameter), and, finally, applies unsigned extension (third parameter) if the access is a fetch.

The field-references given in the table reflect a bias towards fullwords. That is, if ALPHA is a fullword address, then the expressions (ALPHA+4) and (ALPHA+8) are also fullword addresses. This bias is natural for VAX, but it is not essential. An alternative field-reference for component D that does not show this bias follows:

```
(ALPHA+5)<5,19,0>            !No analogue in BLISS--36
```

This field-reference is different from that given previously for D, but it selects the same bits of storage.

Any of the field-references can be used for either a fetch or a store operation. For example, to place the value 7 in component D of ALPHA, write the following:

```
(ALPHA+4)<13,19,0> = 7
```

11.10.3.3. BLOCK Allocation

A specific block data segment is allocated by means of a BLOCK structure-attribute. The attribute provides values for the allocation-formals of the BLOCK structure-declaration. The following declaration allocates storage for the DCB block named ALPHA:

```
OWN
    ALPHA: BLOCK[3,4];
```

The structure-attribute in this example is BLOCK[3,4], and it provides the values 3 and 4 for the allocation-formals N and UNIT, respectively. When storage is allocated for ALPHA, the structure-size expression in the declaration of BLOCK is evaluated. That expression is N*UNIT and its value is therefore 12. Thus 12 bytes of storage (3 fullwords) are allocated for ALPHA.

An equivalent declaration of ALPHA is as follows:

```
OWN
    ALPHA: BLOCK[3];          !Also valid in BLISS--36
```

In this declaration, the structure-attribute does not give a value for UNIT, so the default value is used. This declaration results in the allocation of three fullwords in BLISS–36 also, whereas the prior version would not be valid in that dialect.

Another equivalent declaration is as follows:

```
LITERAL
    DCB_SIZE = 3;
...
OWN
    ALPHA: BLOCK[DCB_SIZE];
```

This example uses a literal-name instead of a numeric-literal to provide the value of the allocation-formal N. This practice is always desirable, and is especially so when ALPHA is one of several data segments of the same form. The use of the name DCB_SIZE tells the reader explicitly that ALPHA will eventually be used for the block diagrammed at the beginning of this section.

11.10.3.4. BLOCK Structure-References

A specific component of a data block is accessed by means of a structure-reference. The structure-reference begins with the name of the data segment and then gives values for the four access-formals of the BLOCK structure declaration.

The following example ends by assigning 7 to component D of ALPHA:

```
LITERAL
    DCB_SIZE = 3;
OWN
    ALPHA: BLOCK[DCB_SIZE];
...
ALPHA[1,13,19,0] = 7;
```

The structure-reference in this example is interpreted as follows:

First, make the following copy of the structure-body of the declaration of BLOCK:

```
(BLOCK+O*UNIT)<P,S,E>
```

Next, replace the formal-name BLOCK with the name ALPHA, providing the following:

```
(ALPHA+O*UNIT)<P,S,E>
```

Next, replace the allocation-formal UNIT with 4, providing the following:

```
(ALPHA+O*4)<P,S,E>
```

Finally, replace the four access-formals, O, P, S, and E, with the corresponding access-actual parameters 1, 13, 19, and 0, providing the following:

```
(ALPHA+4)<1,13,19,0>
```

This is the same as the field-reference given for component D in *Section 11.10.3.2, "BLOCK Field-References"*.

11.10.3.5. BLOCK Field-Declarations

The reference to component D of ALPHA is improved by the use of the BLOCK structure-name, but it still requires a list of integer parameters, [1,13,19,0], that bears no obvious relation to the description "component D of DCB".

You could solve this problem by defining a macro, such as the following:

```
MACRO
    DCB_D = 1,13,19,0 %;
```

However, BLISS provides a special feature, the *field-declaration*, for this purpose.

The following program fragment shows the complete mechanism for handling the block ALPHA:

```
LITERAL
    DCB_SIZE = 3;
FIELD
    DCB_FIELDS =
        SET
        DCB_A = [0,0,32,0],
        DCB_B = [1,0,8,0],
        DCB_C = [1,8,5,0],
        DCB_D = [1,13,19,0],
        DCB_E = [2,0,32,0]
        TES;
MACRO
    DCB = BLOCK[DCB_SIZE] FIELD(DCB_FIELDS) %;
OWN
    ALPHA: DCB;
...
ALPHA[DCB_D] = 7;
```

The field-declaration defines the four-integer code for each component and also gives a name, DCB_FIELDS, to the five field-names thus declared.

The declaration of the macro-name DCB is the final convenience; it permits the block layout that is associated with ALPHA to be designated by a single name, DCB.

When the macro-call on DCB is expanded, the declaration of ALPHA becomes the following:

```
OWN
    ALPHA: BLOCK[DCB_SIZE] FIELD(DCB_FIELDS);
```

The field-attribute allows the five field-names associated with DCB_FIELDS to be used in structure-references for ALPHA.

11.10.4. BLOCKVECTOR Structures

A BLOCKVECTOR structure is a vector of blocks. The number of elements (n) is the extent of the vector, and the size of each element is the size of a single block. The elements are numbered from 0 to n–1. The structure-declaration for BLOCKVECTOR in each dialect is as follows:

- In BLISS–16:

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

- In BLISS–32:

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

- In BLISS–36:

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

The formal names of the structure-declaration have the following meanings:

Formal Name	Meaning
I	The number of the block element. Valid values are 0 through n-1.
O	The offset to a field. Valid values are 0 through BS-1.
P	Bit offset from the addressable-unit to the beginning of the field.
S	Size of the field in bits. Valid values are 0 through %BPVAL.
E	Extension rule. Valid values are 0 for zero-extension and 1 for sign-extension.
N	The number of block elements in the vector.
BS	The number of allocation-units in each block element.
UNIT	The number of addressable-units in the allocation-unit.

The BLOCKVECTOR structure is a combination of the allocation and access definitions from the BLOCK and VECTOR structures.

Using this structure, a declaration of a vector of DCB blocks (used as an example of the BLOCK structure in *Section 11.10.3, "BLOCK Structures"*) is written as follows:

```
OWN XXX: BLOCKVECTOR[100,DCB_SIZE] FIELD(DCB_FIELDS);
```

This declaration allocates storage for 100 DCB blocks, each of which is three fullwords in size. If the contents of a variable J is 2, then the following fetches the value of the D field of the third block in the vector:

```
.XXX[.J,DCB_D]
```

Observe that the same field-declaration used with the block discussed in *Section 11.10.3, "BLOCK Structures"* is used with the block vector discussed here.

11.11. Other Structures

The predeclared structures described in the previous section are included in BLISS because they occur frequently in many types of programs. However, they are only a sample of the wide range of structures that can be defined with the structure declaration. This section describes additional structures that illustrate some other possibilities.

To minimize the complexity of the example structures presented, only fullword versions of the structures are defined. These examples could be augmented in a variety of ways to be more flexible. Also, the structure-declarations are written in parameterized, transportable form (using the predeclared literal %UPVAL) so that they are valid in all dialects.

11.11.1. "One-Origin" Vector Structures

The definition of vector presented previously numbered the elements of the vector (n) from 0 to n–1. In some applications, it is more natural to number the elements from 1 to n instead.

A structure that accomplishes this is as follows:

```
STRUCTURE
    VECTOR1[I; N] =
        [N*%UPVAL]
        (VECTOR1+(I-1)*%UPVAL);
```

This structure differs from the VECTOR structure previously presented in that 1 is subtracted from the element number before the offset relative to the base of the vector is computed.

11.11.2. "Bounds Checking" Vector Structures

On occasion, particularly during debugging, it is desirable to perform validity checking of the access-actuals of a structure-reference. For the VECTOR1 structure just given, bounds checking can be accomplished as follows:

```
STRUCTURE
    VECTOR1CHK[I; N] =
        [N*%UPVAL]
        BEGIN
        LOCAL T;
        T = I;
        IF .T LSS 1 OR .T GTR N
        THEN
             BEGIN
             ERROR(.T);
             T = 1;
             END;
        VECTOR1CHK+(.T-1)*%UPVAL
        END;
```

This structure calls a routine ERROR for those cases in which the value of I is not in the valid range of 1 through N inclusive.

11.11.3. Two-Dimensional Array Structures

A zero-origin two dimensional array structure can be defined as follows:

```
STRUCTURE
    ARRAY[I, J; M, N] =
        [M*N*%UPVAL]
        (ARRAY+(I*N+J)*%UPVAL);
```

This structure stores elements in row order as in PL/I.

A similar structure that stores elements in one-origin column order, as in FORTRAN, can be defined as follows:

```
STRUCTURE
    ARRAYBYCOL[I, J; M, N] =
        [M*N*%UPVAL]
        (ARRAY+((J-1)*M+(I-1))*%UPVAL);
```

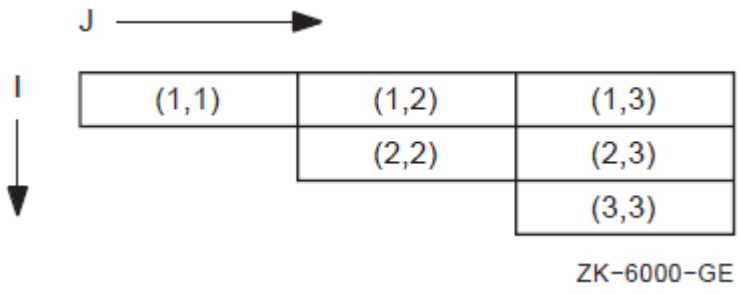
This structure differs from the previous example in the following ways:

- I is replaced by I–1 and J is replaced by J–1 to get one-origin numbering of the elements.
- I and J are interchanged in the structure-body, as are M and N, to get column ordering instead of row ordering.

11.11.4. Symmetric Array Structures

A symmetric array is a square array in which the contents of A[I,J] is equal to the contents of A[J,I]. For such an array, it is not necessary to allocate storage for the entire array.

A symmetric 3-by-3 array can be diagrammed as follows:



The number of elements needed to represent a symmetric array is as follows:

```
n * (n+1)/2
```

where *n* is the number of elements in each dimension. In the 3-by-3 example above, this gives 3*4/2, or 6, elements.

The storage for such an array can be allocated with the elements in the following order:

```
(1,1), (1,2), (2,2), (1,3), (2,3), (3,3)
```

If j is greater than or equal to i, then the linear position of the (i,j) element in the storage sequence is given by the following formula:

```
j*(j-1)/2+i
```

In the 3-by-3 example above, the position of the (2,3) element is as follows:

```
3*(3-1)/2+2 = 5
```

That is, element (2,3) is the fifth element of the linear sequence.

This analysis can be incorporated into a structure declaration for symmetric arrays as follows:

```
STRUCTURE
    SYMARRAY[I, J; M] =
        [(M*(M+1)/2)*%UPVAL]
        (SYMARRAY-%UPVAL+
            (IF J GTR I
            THEN
                J*(J-1)/2+I
            ELSE
                I*(I-1)/2**
            )*%UPVAL
        );
```

Declaration and use of this structure is the same as for an ordinary two-dimensional one-origin array. For example:

```
OWN SYMX: SYMARRAY[10,10];
```

This declares and allocates a 10-by-10 symmetric array named SYMX. It occupies 55 fullwords of storage.

The sum of the 100 logical elements of the array can be computed as follows:

```
SUM = 0;
INCR I FROM 1 TO 10 DO
    INCR J FROM 1 TO 10 DO
        SUM = .SUM + .SYMX[.I,.J];
```

11.11.5. Noncontinuous Block Structures

The predeclared definition of the BLOCK structure given previously assumes that all of the fields of the block are contiguous in memory. In some cases this might not be possible or desirable. For example, a storage management subsystem might be in use that provides only a fixed-size block of memory. In such a circumstance it may still be desirable to reference a "logical block" as an entity even though it might be represented using more than one physical block of memory.

The following structure illustrates a way to achieve this:

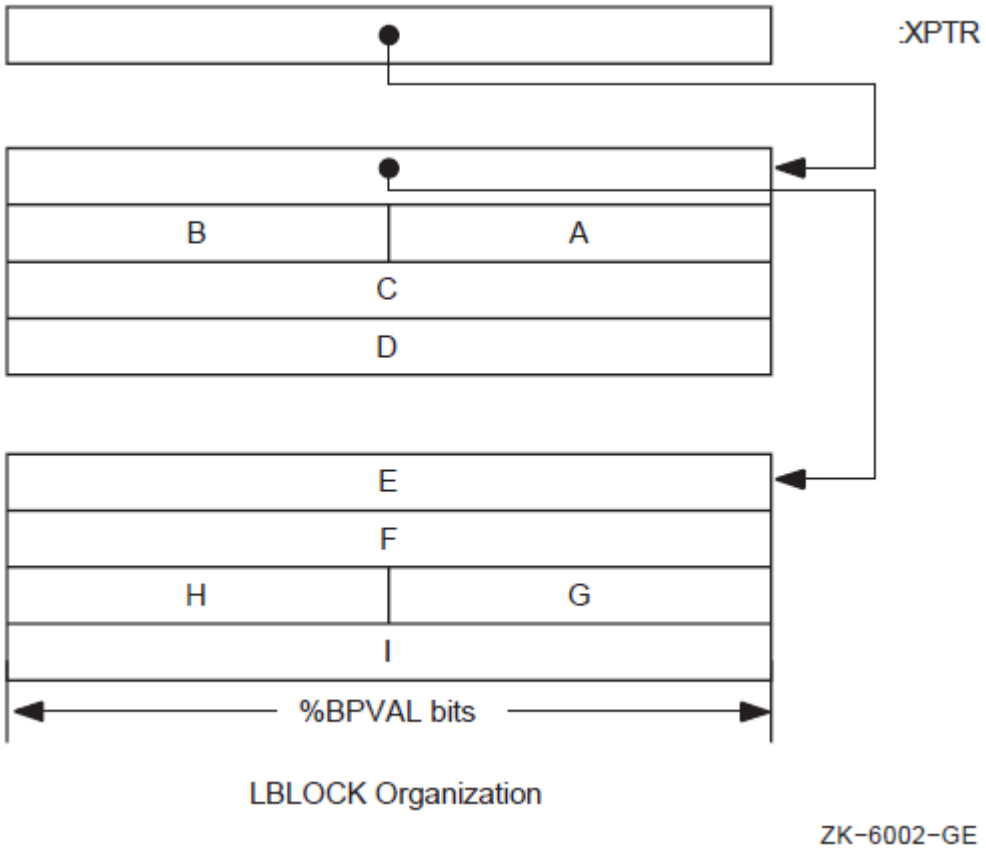
```
STRUCTURE
    LBLOCK[O, P, S, E, I] =
        (CASE I FROM 0 TO 1 OF
            SET
            [0]: (LBLOCK+O*%UPVAL);
            [1]: (.LBLOCK+O*%UPVAL);
            TES
        )<P,S,E>;
```

Because this structure is only intended to be used with dynamically allocated memory, the definition does not contain a structure-size expression.

A typical declaration of a data segment that points to an instance of this structure is as follows:

```
OWN XPTR: REF LBLOCK;
```

To understand this structure, consider the following diagram:



LBLOCK Organization

ZK-6002-GE

The diagram illustrates a logical block consisting of 9 fields named A through I. The logical block is represented as two physical blocks. Each physical block consists of four fullwords, the assumed fixed-size storage management unit. The arrows indicate fields that contain the address of the first block and of the remainder of the logical block.

The first physical block is like the BLOCK structure described in *Section 11.10.3, "BLOCK Structures"*. However, the access formal list for the LBLOCK structure includes an additional formal name, I, that the BLOCK structure did not have. This formal name is used in the structure-body to choose one of two expressions as the structure address expression.

The field-name for A is defined as follows:

```
FIELD A = [1,0,%BPVAL/2,1,0];
```

When used in a structure-reference to XPTR, the last 0 in this definition causes the first case-line of the structure-body to be used, and thus the following reference is like a BLOCK reference:

```
XPTR[A]
```

A field in the second physical block, such as F, is defined with a 1 as the last value, as in the following:

```
FIELD F = [1,0,%BPVAL,1,1];
```

The last 1 in this definition causes the second case-line to be used. Examination of the second case-line shows that it is just like the first except that the contents of the first fullword of the first physical block is used as the base for applying the offset, position, size, and extension values.

A reference to this field is written in the same way as a reference to the A field:

```
XPTR[F]
```

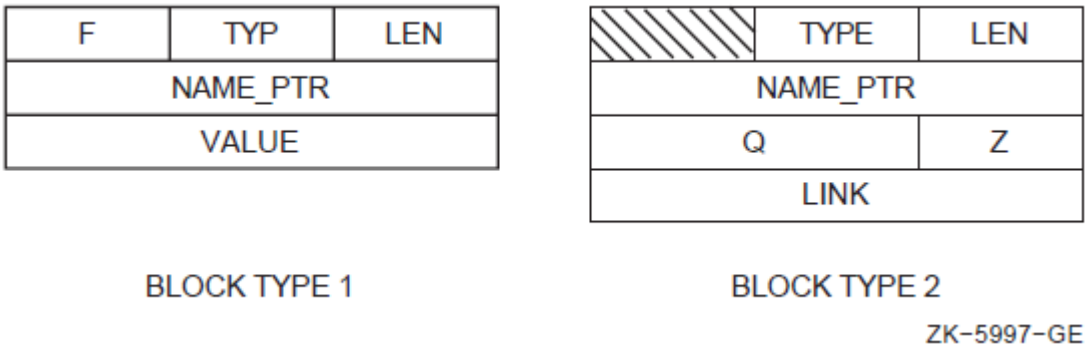
The extra indirection used to reference this field is hidden in the structure and field definitions used to define the logical structure.

11.11.6. Partially Overlaid Structures

Some programming applications require data structures that are similar with respect to some, but not all, of their fields.

For example, consider the symbol table of a compiler. The table must accommodate different kinds of identifiers (symbols), and has a different kind of block for each kind of identifier. However, in order to make the table useful, some fields will appear in all blocks of the table. One such common field will be the *type* field, which specifies which kind of identifier a given block represents.

As another example, consider the table of device control blocks in an operating system. Once again, the table must have different kinds of blocks, one kind for each kind of device; and, once again, some fields will appear in all blocks of the table. In this example, the common fields might be the priority level, a pointer to a queue of operations, and a device type code. For example:



The diagram shows two different blocks that share some common fields, namely: LEN, TYP, and NAME_PTR. Each block also has fields that are not common with the other block; indeed, the blocks are not even the same size.

The following declarations illustrate one way to code the definitions of these two blocks, using BLISS–36 as the sample dialect:

```
FIELD
    COM_FLDS =
        SET
        LEN = [0,0,12,0],
        TYP = [0,12,12,0],
        NAME_PTR = [1,0,36,0]
        TES,

    TYP1_FLDS =
        SET
```

```
        F = [0,24,12,0],
        VALUE = [2,0,36,0]
        TES,

    TYP2_FLDS =
        SET
        Z = [2,0,18,0],
        Q = [2,18,18,1],
        LINK = [3,0,36,0]
        TES;

MACRO
TYP1_BLOCK = BLOCK[3] FIELD(COM_FLDS,TYP1_FLDS) %,
TYP2_BLOCK = BLOCK[4] FIELD(COM_FLDS,TYP2_FLDS) %;
```

The field-declaration defines three sets of fields:

COM_FLDS	For fields that are common to both types of block.
TYP1_FLDS	For fields that are specific to the first type of block.
TYP2_FLDS	For fields that are specific to the second type of block.

The macro-declaration defines two macros, one for each kind of block; the expansions give the attributes appropriate for each kind of block.

These macro-names can be used in data declarations such as the following:

```
OWN
    STARTUP: TYP1_BLOCK;
LOCAL
    PTR: REF TYP2_BLOCK;
```

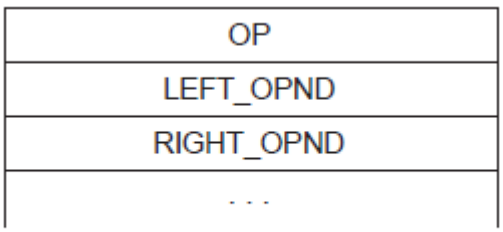
Observe that in the declaration of PTR (as LOCAL) the structure-attribute is REF BLOCK[4], where REF is given explicitly and BLOCK[4] results from the expansion of TYP2_BLOCK. If BLOCK[4] and FIELD (COM_FLDS,TYP2_FLDS) had been given in the opposite order in the macro definition of TYP2_BLOCK, then additional macro definitions would be needed in order to declare data segments with REF structure-attributes. The definition technique shown above has two advantages:

- The common definition information is given only once, thereby avoiding the possibility of clerical errors in giving the same information in multiple field-set definitions.
- Depending on specific details, changes or additions to the common fields can be made in one place, which is easier and more reliable than making corresponding changes in many places.

11.11.7. General-Purpose Structures for Default Structure References

Some programming applications involve complicated data structures using blocks of various types connected together by pointers. If the nature of the application involves frequent access to blocks related to a given block by "following pointers", there may well be notational advantages to using a default structure (see *Section 11.8, "Default-Structure-References"* and *Section 18.2, "Switches-Declarations"*).

For example, suppose the following block is being used to represent a node in a tree structure, such as might be used for expressions in a compiler.



The op field is used to contain a code for the kind of arithmetic operator represented, and the LEFT_OPND and RIGHT_OPND fields are used to contain addresses of other such nodes.

A routine to compare the OP fields of the two subnodes of a given node for equality might be written as follows:

```
ROUTINE COMPARE_SUBOPS(NODE) =
     BEGIN
     MAP NODE: REF TREE FIELD(TREE_FIELDS);
     LOCAL
          L_PTR: REF TREE FIELD(TREE_FIELDS),
          R_PTR: REF TREE FIELD(TREE_FIELDS);
     L_PTR = .NODE[LEFT_OPND];
     R_PTR = .NODE[RIGHT_OPND];
     IF .L_PTR[OP] EQL .R_PTR[OP]
     THEN
         ...;   ! Actions if subnodes have same OP value
     END;
```

The structure and field name definitions assumed in this example should be obvious from earlier examples and are not shown. You can achieve the same effect using a default structure as follows:

```
ROUTINE COMPARE_SUBOPS1(NODE) =
     BEGIN
     SWITCHES STRUCTURE(REF TREE);
     IF .NODE[LEFT_OPND][OP] EQL .NODE[RIGHT_OPND][OP]
     THEN
          ...;   ! Actions if subnodes have same OP value
     END;
```

This second version is slightly shorter. It is also more suggestive of the "logical" access being performed because intermediate assignments are not needed simply to obtain a data segment name (such as L_PTR in the first version) that is declared with the appropriate structure properties for each step along the path of access.

Observe that the default structure in this example is a REF structure. This means that each step in the access path necessarily makes a fetch to obtain the base address for the next field access.

Chapter 12. Routines

Routines are the logical units from which a program is built. Each routine describes a portion of the program that is relatively complete and independent. BLISS permits a routine to have its own block structure and local data.

A program has a single main routine (see *Section 19.2, "Module-Switches"*). The main routine controls the computation, but it can delegate parts of the computation to subordinate routines. Each subordinate routine can, in turn, delegate part of its computation to its own subordinate routines. A routine can also call an external routine (one defined outside of its own block or module) to perform a commonly needed function, for example.

The first two sections of this chapter describe routine-calls. The remaining five sections describe routine-declarations.

The linkage-declaration, which controls the instruction sequence generated for a call on a given routine, and the register-management discipline used within the routine, is described in *Chapter 13, "Linkages"*.

12.1. Ordinary-Routine-Calls

A routine-call causes the execution of a routine that has been declared as part of the same module or some other BLISS module, or of a program written in another language. Two kinds of routine-calls are provided: ordinary and general. The ordinary-routine-call is the most commonly used form: it gives the name of a routine and relies on the compiler to determine, from the declaration of the named routine, the appropriate linkage (or calling sequence).

A general-routine-call is self-contained. It gives all of the information needed for calling the routine.

The following is an example of an ordinary-routine-call:

```
OWN
    A,
    B;
EXTERNAL ROUTINE
    RFACT;
.
.
.
A = RFACT(.B)
END
```

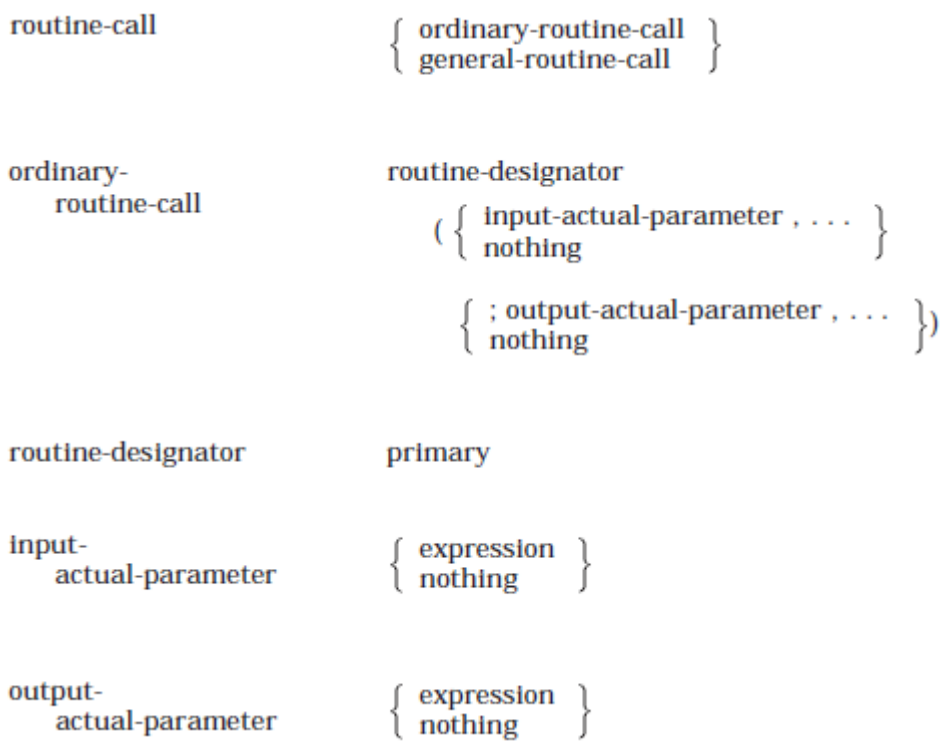
The RFACT routine is declared in another module. The function of the routine is to determine the factorial of a given parameter. The result is the value of the routine; therefore, the routine does not have a NOVALUE attribute. The routine-call RFACT(.B) causes the contents of input-actual-parameter B to be passed to the factorial routine and the returned result to be assigned to location A. The declaration of routine RFACT is given in *Section 12.4, "Ordinary-Routine-Declarations"*.

In the example, the routine-call is used to pass an input-parameter; however, output-parameters can also be passed. When this is done, each output-actual-parameter is treated like the left-hand side of an assignment expression defining where an output-register value (from the called routine) is to be stored.

Output-parameters permit a routine to return results that are larger than a BLISS value or to return several values at once. For example, a double-precision floating point value can be returned in R0 and R1.

In the routine-call syntax, output-parameters follow input-parameters and are separated by a semicolon (;).

12.1.1. Syntax



12.1.2. Restrictions

The number of input-actual-parameters in a routine-call must agree with the number of input-formal-parameters in the routine-declaration. This restriction can be relaxed through use of the linkage-functions described in *Section 13.6, "Linkage-Functions"*.

The value of each input-actual-parameter must be consistent with the context in which the corresponding input-formal-parameter is used in the routine declaration.

An output-actual-parameter can be any expression, including an undotted register-name qualified by position, size, and sign extension information (that is, a field-reference).

The number of output-actual-parameters must be less than or equal to the number of output-formal-parameters specified in the routine declaration.

An output-actual-parameter must not be specified if a corresponding output-parameter-location register is not specified in the linkage. The evaluation of the routine-designator must yield the value of a name that has been declared ROUTINE.

The linkage of the routine-designator (determined as described in *Section 12.1.3, "Semantics"* must be the same as the linkage-attribute in the declaration of the routine that is called.

A linkage-name defined with the linkage-type INTERRUPT or RSX_AST must not be used in a general-routine-call.

The order in which the routine-designator and actual-parameters are evaluated is as follows: Input-actual-parameters are evaluated before the routine call, and output-actual-parameters are evaluated when the routine returns to the caller.

12.1.3. Semantics

An ordinary-routine-call is interpreted as follows:

1. Evaluate the routine-designator and the actual-parameters.

2. Determine the linkage to be used with the routine-designator. If the routine-designator is a routine-name, then the linkage is given by the linkage-attribute (explicit or default) in the declaration of the routine-name. Otherwise, the linkage is given by the linkage-name established in a LINKAGE switch or, if no LINKAGE switch applies, the linkage is the default linkage-name for the dialect in use (BLISS for BLISS–16/32; BLISS36C for BLISS–36).

3. Associate the actual-parameters with the formal-parameters of the routine called. The value of the ith actual-parameter becomes the content of the ith formal-parameter.

4. Create a stack frame. The kind of stack frame and the details of its organization depend on the linkage of the routine.

5. Evaluate the routine-body.

6. Delete the stack frame.

7. Evaluate the output-actual-parameter expressions and assign the returned output-register values to the appropriate output-actual-parameters.

8. If a value is returned, use that value as the value of the routine-call.

The linkage used in a routine-call does not affect the semantics of the call, but instead affects the details of how the call is carried out. Linkages are described in *Chapter 13, "Linkages"*.

12.1.4. Pragmatics

An input-actual-parameter in a routine-call can be a %REF standard function. This function is especially designed for use in routine-calls. It is described and illustrated in *Section 5.2.2.3, "The %REF Function"*.

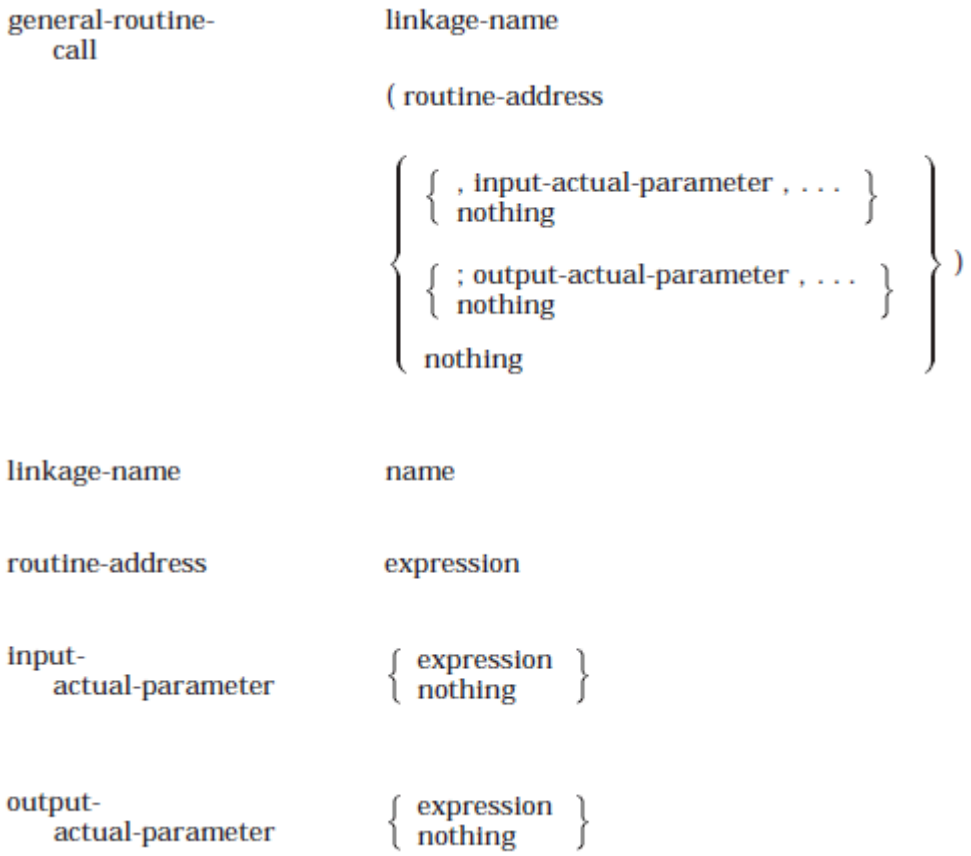
12.2. General-Routine-Calls

A routine whose address is computed during execution can be called with a linkage other than the default linkage using a general-routine-call. The following is an example of a general-routine-call:

```
EXTERNAL ROUTINE
    F1: FORTRAN_SUB NOVALUE,
    F2: FORTRAN_SUB NOVALUE,
    F3: FORTRAN_SUB NOVALUE;
BIND
    TABLE = UPLIT(F1,F2,F3) : VECTOR;
...
FORTRAN_SUB(.TABLE[.I], P1, P2)
...
```

The address of the FORTRAN routine to be called is computed by fetching an element of a vector. Because the routine has linkage-type FORTRAN_SUB, the general-routine-call must be used to give the compiler the information necessary to generate the correct form of routine-call.

12.2.1. Syntax



12.2.2. Restrictions

BLISS–16 ONLY

A linkage-name defined with the linkage-type INTERRUPT or RSX_AST must not be used in a general-routine-call.

The evaluation of the routine-address expression must yield the address of a routine that is declared with the specified linkage-name as its linkage-attribute.

The number of input-actual-parameters in a routine-call must agree with the number of input-formal-parameters in the routine-declaration. This restriction can be relaxed through use of the linkage-functions described in *Section 13.6, "Linkage-Functions"*.

The value of each input-actual-parameter must be consistent with the context in which the corresponding input-formal-parameter is used in the routine-declaration.

An output-actual-parameter can be any expression, including an undotted register-name qualified by position, size, and sign extension information (that is, a field-reference).

The number of output-actual-parameters must be less than or equal to the number of output-formal-parameters specified in the routine declaration.

An output-actual-parameter must not be specified if a corresponding output-parameter-location register is not specified in the linkage.

The order in which the routine-address expression and actual-parameters are evaluated is as follows: Input-actual-parameters are evaluated prior to the routine call, and output-actual-parameters are evaluated when the routine returns to the caller.

12.2.3. Semantics

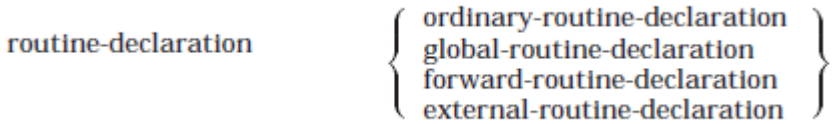
In a general-routine-call, the routine-address expression is interpreted as the address of the routine to be called, and the remaining expressions are interpreted as the actual parameters of the call. The linkage to be used is given by the linkage-name. In all other respects, the semantics is the same as for an ordinary-routine-call.

12.3. Routine-Declarations

A routine-name can be declared in five different ways in BLISS. An *ordinary*-routine-declaration is used to give the definition of a routine that is used only in the block in which it is declared. A *global*-routine-declaration is used to give the definition of a routine that is used in other modules as well as in the module in which it is declared. A *forward*-routine-declaration declares the name of a routine so that it can be called from a point in the block that precedes its complete definition, which is given by an ordinary- or global-routine-declaration. An *external*-routine-declaration declares the name of a routine whose definition is given as a global-routine-declaration in another module. A *bind*-routine-declaration gives the definition of the address of a routine in terms of an expression.

The first four ways of declaring a routine-name are described in the following sections. The bind-routine-declaration is described in *Section 14.4, "Bind-Routine-Declarations"*.

12.3.1. Syntax



12.3.2. Semantics

The semantics of the routine-declaration is given in the following sections where each kind of routine-declaration is considered separately.

12.4. Ordinary-Routine-Declarations

An ordinary-routine-declaration defines a routine. The scope of the declared *routine-name* is the immediately containing block (including all contained blocks). The declaration includes an expression, the *routine-body*, which is evaluated each time the routine is called. The declaration also includes a list

of *formal-names*. When the routine is called, the value of each actual-parameter in the routine-call is assigned to the corresponding formal-name. The formal-names can be accessed in the routine-body as if they were LOCAL data segment names, except that values must not be assigned to them.

A BLISS routine can be *recursive*. A routine is recursive if it can be called while a previous call to the routine is still active. Recursion can be *direct* or *indirect*. Direct recursion occurs when the routine contains a call on itself; for example, the routine-body for the routine A contains a call on the routine A. Indirect recursion occurs when the routine contains a call on another routine, which ultimately results in a call on the routine being declared; for example, the routine-body for the routine A contains a call on the routine B, which contains a call on the routine A.

An example of an ordinary-routine-declaration follows:

```
ROUTINE AVERAGE3(F1,F2,F3) = (.F1 + .F2 + .F3)/3;
```

The routine `AVERAGE3` has three formal-names (`F1`, `F2`, and `F3`). An example of a call on this routine follows:

```
AVERAGE3(5, .A, .B*.C)
```

Another example of an ordinary-routine-declaration is the declaration of a factorial routine. This routine computes the mathematical function factorial(n):

```
ROUTINE IFACT (N) =
    BEGIN
    LOCAL
        RESULT;
    RESULT = 1;
    INCR I FROM 2 TO .N DO
        RESULT = .RESULT*.I;
    .RESULT
    END;
```

When the routine IFACT is called, it computes the factorial of the actual-parameter specified. If the content of N is less than 2, the indexed-loop is not executed and the value of the routine is 1. An example of a call in this routine follows:

```
IFACT(.A * .B)
```

In this example, if the content of `A` is assumed to be 2 and the content of `B` is assumed to be 3, the result returned by the call is 720.

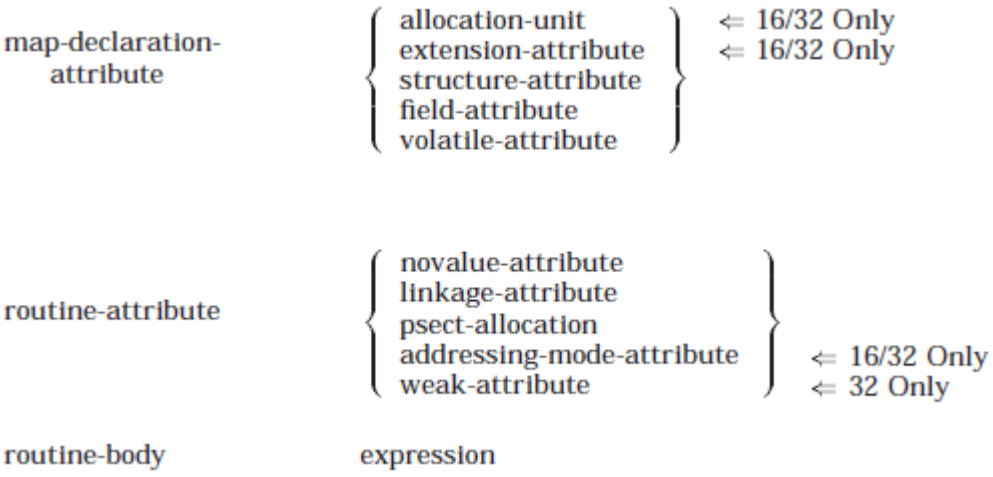
The factorial routine could be rewritten as a directly recursive routine, as follows:

```
ROUTINE RFACT (N) =
   IF .N GTR 1
       THEN
           .N * RFACT (.N - 1)
       ELSE
           1;
```

For the computation of a factorial the first version, IFACT, is more efficient than the recursive version, RFACT. Recursion is used when it is the most natural or efficient method.

12.4.1. Syntax

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

routine-definition       routine-name

                         { ( input-list )
                         { ( ; output-list )
                         { ( input-list ; output-list )
                         { nothing

                         { : routine-attribute . . . }
                         { nothing                   }

                         = routine-body

routine-name             name

input-list               input-formal-parameter , . . .

output-list              output-formal-parameter , . . .

input-
    formal-parameter
output-
    formal-parameter     formal-item

formal-item              formal-name { : formal-attribute-list }
                                     { nothing                }

formal-name              name

formal-
    attribute-list       { map-declaration-attribute . . . }
```



12.4.2. Restrictions

The number of input-formal-parameters in the routine-declaration must agree with the number of input-actual-parameters in the routine-call. This restriction can be relaxed through use of the linkage-functions described in *Section 13.6, "Linkage-Functions"*.

The number of output-formal-parameters in the routine-declaration must be less than or equal to the number of output-parameter-locations specified in the linkage-declaration.

An output-formal-parameter must not be specified if a corresponding output-parameter-location is not specified in the linkage.

The value of an output-formal-parameter is undefined until it is assigned a value within the routine-body.

An input-formal-name must not be assigned a value.

Both the value of a formal-name and its content are undefined except during the evaluation of the routine-body.

If the routine is declared with the NOVALUE attribute, it must not be called in a context that requires a value and if any RETURN expression in the routine-body has a returned-value, the expression is evaluated but its value is not used. If the routine does not have the NOVALUE attribute, any RETURN expression in the routine-body as well as the routine-body itself must have a returned-value.

Suppose the routine-body of a given routine, routine A, contains the declaration of another routine, routine B. If a name is a formal-name for routine A, then that name cannot be used as such within routine B. Such usage would be an "up-level" reference, which is prohibited for formal-names just as for local-names (see *Section 10.5, "Local-Declarations"*).

12.4.3. Defaults

Each formal-name is implicitly declared by a routine-declaration. Each declaration is assumed to be a scalar, with a default allocation-unit and extension-attribute (BLISS–16/32 only). If this assumption is not appropriate, other map-declaration-attributes can be specified (see *Section 12.4.5.3, "Attributes for Formal-Names"*).

If a linkage-attribute is not given and the routine is in the scope of a LINKAGE switch, then the default linkage-attribute is the linkage-name given by the LINKAGE switch (see *Section 18.2, "Switches-Declarations"* and *Section 19.2, "Module-Switches"*). Otherwise, the default is the predeclared linkage-name BLISS for BLISS–16/32, or BLISS36C for BLISS–36.

12.4.4. Semantics

The compiler makes use of the information in an ordinary-routine-declaration as follows:

1. The attributes and keywords are processed.

2. The routine-body is processed. Input- and output-formal-names are treated as local variable names that are declared in an implicit block enclosing the routine-body. The input-formal-names are then initialized with the values of the corresponding input-actual-parameters from a routine-call; however, output-formal-names are not initialized with corresponding output-actual values.

3. When the routine returns to the caller, the contents of the data-segment associated with each output-formal-parameter, are moved to the registers specified in the associated linkage-declaration.

4. If the routine is declared with the NOVALUE attribute, the mechanism for returning a value is suppressed.

12.4.5. Pragmatics

The following sections give examples that illustrate various aspects of the routine facility of BLISS.

12.4.5.1. Parameter Passing

The value of each actual-parameter of a routine-call is passed to the routine by means of the corresponding formal-name. However, the value of the formal-name is not the value of the actual-parameter. Instead, each formal-name designates a data segment that contains the value of the actual parameter. The data segment designated by the formal-name is defined only during evaluation of the routine-body, and it is "temporary" in that sense.

Because it is the value of an actual-parameter that is normally of interest (rather than the address of the temporary data segment that contains that value), a use of a formal-name without a preceding fetch-operator is often an error. For example:

```
ROUTINE AVERAGE3(F1,F2,F3) =
    (.F1 + .F2 + .F3)/3;
```

This routine is called with three actual-parameters whose values are to be averaged. An example of a call on the routine follows:

```
AVERAGE3(5, .A, .B*.C)
```

Each formal-name of the routine can be thought of as a special kind of LOCAL name that is declared in the implicit block that surrounds the routine-body. Therefore, the routine-body for AVERAGE3 can be thought of as the following block:

```
BEGIN
LOCAL
    F1,
    F2,
    F3;
F1 = 5;
F2 = .A;
F3 = .B*.C;
(.F1 + .F2 + .F3)/3
END
```

This interpretation shows that it is .F1, .F2, and .F3 that represent the values to be averaged, not F1, F2, and F3.

In the preceding example, the routine-call supplied values that were intended for calculation. It is also possible for a routine-call to supply values that are intended for use as addresses. For example:

```
ROUTINE EXCHANGE(X,Y): NOVALUE =
    BEGIN
    LOCAL TEMP;
    TEMP = ..X;
    .X = ..Y;
    .Y = .TEMP;
    END;
```

This routine is called with two actual-parameters whose values are the addresses of data segments. An example of a call on the routine follows:

```
EXCHANGE(Q,R)
```

When this call is evaluated, the contents of Q and R are interchanged. Once again, each formal-name can be thought of as a special kind of LOCAL name. Thus the given parameters Q and R are represented by .X and .Y, respectively, not by X and Y.

Note that routines to be called from FORTRAN must assume that actual-parameter values are always the addresses of data segments. This is so because FORTRAN routines pass parameters by address, not by value.

For example:

```
ROUTINE AVERAGE3A(F1,F2,F3) =
    (..F1 + ..F2 + ..F3)/3;
```

This routine requires that the actual-parameters be the addresses of the values to be averaged. Thus a BLISS call on this routine might be as follows:

```
AVERAGE3A(UPLIT(5), A, %REF(.B*.C))
```

This call on AVERAGE3A gives the same value as the call, given earlier, on AVERAGE3. The first actual-parameter uses a UPLIT (see *Section 4.4, "PLITs"*) to supply the address of the numeric-literal 5. The second actual-parameter simply uses the name A (without a fetch operator) to get the address of the value. The third actual-parameter uses the %REF standard function (see *Section 5.2, "Executable-Functions"*) to supply an address for the value of the expression .B*.C.

The routine AVERAGE3A uses addresses of values where values would have been sufficient for interaction with other BLISS routines. That is to say, it does not minimize indirection. However, the routine is valid and, written in this way, can be made callable from programs written in the FORTRAN language by the addition of the FORTRAN_FUNC linkage-attribute (see *Section 13.5, "Common Predeclared Linkage-Names"*).

12.4.5.2. Allocation of Formal-Name Data Segments

While data segments for formal-names are like local data segments in most respects (as discussed in *Section 12.4.5.1, "Parameter Passing"*), they are not necessarily allocated in the same way as local data segments. Formal data segments are allocated and assigned values by the routine making a call, rather than by the routine that is called. The calling routine can arrange to allocate formals in static memory that is protected from write access rather than, for example, in a temporary segment in a stack frame. This is an optimization because, under suitable conditions, the calling routine does not need to allocate and assign values for the formals each time the call is made. Moreover, the calling routine can even

be able to use the same formal data segments for different routine calls if they have the same number and sequence of actual parameter values. A restriction given in *Section 12.4.2, "Restrictions"*, namely, that a formal name must not be assigned a value, assures that it is valid for a calling routine to use such optimizations.

12.4.5.3. Attributes for Formal-Names

If the default attributes (UNSIGNED WORD in BLISS–16, UNSIGNED LONG in BLISS–32, none in BLISS–36) are not appropriate for a formal-name, an appropriate attribute can be selected from the map-declaration-attributes. An example of the use of a structure-attribute in an ordinary routine declaration is as follows:

```
ROUTINE ZEROBIT(A : REF BITVECTOR[12],B,C): NOVALUE =
    BEGIN
    IF .A[.B]
    THEN
        BEGIN
        A[.B] = 0;
        .C = ..C + 1;
        END;
    END;
```

The structure-attribute `REF BITVECTOR[12]` is provided for the first formal-name (`A`). Assuming the content of `B` is i, the routine ZEROBIT tests the ith bit of the bit vector structure `A`. If that bit is 1, it is set to 0 and the content of the location pointed to by `.C` is incremented.

12.4.5.4. Computed Routine Addresses

A routine-call usually begins with a routine-name, which designates the routine in an explicit and constant way. However, a routine-call can begin with any expression that yields a valid routine address. For example:

```
ROUTINE ENTVAL(A,ERR): NOVALUE =
    BEGIN
    ... !Try to enter .A in LIST1
    IF .FILLED THEN (.ERR)(1, .A);
    ... !Try to enter .A in LIST2
    IF .FILLED THEN (.ERR)(2, .A);
    END;
```

Assume that this routine tries to put the content of A into two lists, LIST1 and LIST2. If the list is filled up, an error message must be printed. However, ENTVAL does not print a message and does not even call a specific routine to print an error message. Instead, ENTVAL calls a routine whose address is given as one of the formal-names.

An example of the use of ENTVAL follows:

```
ROUTINE ERRX(N,VAL): NOVALUE =
    BEGIN
    ...             !Print error message for invalid .X
    END
.
.
.
ENTVAL(.X,ERRX)
```

In this example, ENTVAL is called to enter the contents of X in the lists. The second parameter of the call is ERRX, which is the name of a routine designed especially to report an invalid value

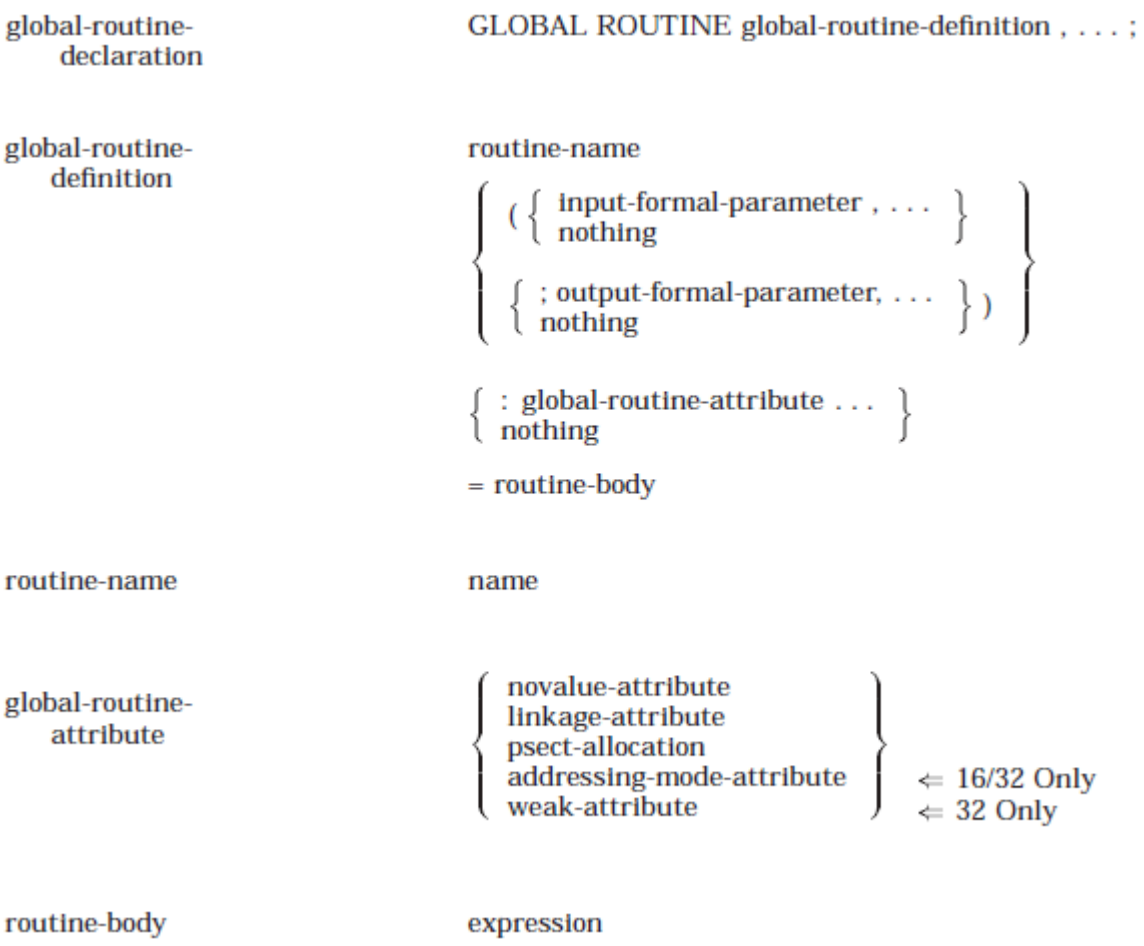
of .X. Observe that the name ERRX in this call does not call the routine ERRX because there are no parentheses following it. Thus, ERRX is not a routine call. Presumably, the same program contains other calls on ENTVAL, and different calls use different routines to report an invalid value.

12.5. Global-Routine-Declarations

A global-routine-declaration provides the same information as the ordinary-routine-declaration. The only difference between these two declarations is their scope. A routine that is declared in an ordinary-routine-declaration can only be called in the block in which the declaration is given (*Section 8.2.4, "Discussion"*). A routine that is declared in a global-routine-declaration can be called outside the block in which it is declared. The scope of the routine-name is extended beyond the block by means of one or more external-routine-declarations in other blocks or modules.

The only differences between the syntax of the ordinary-routine-declaration and the global-routine-declaration are that the GLOBAL keyword is required in the latter and, in BLISS–32 only, the weak-attribute is permitted in a global-routine-declaration.

12.5.1. Syntax



12.5.2. Restrictions

The restrictions given in *Section 12.4.2, "Restrictions"* for ordinary-routine-declarations also apply to global-routine-declarations.

BLISS–16 and BLISS–36 restrictions on names declared as global are given in *Section 4.5.2, "Restrictions"*.

12.5.3. Defaults

The defaults given in *Section 12.4.3, "Defaults"* for ordinary-routine-declarations also apply to global-routine-declarations.

12.5.4. Semantics

The compiler makes use of the information in a global-routine-declaration as follows:

1. The global nature of the routine is recorded. An indicator is set for the linker to show that this is a global-declaration. If the routine-declaration has the weak-attribute, another indicator is set for the linker.

2. The semantics are then the same as the semantics for an ordinary-routine-declaration, given in *Section 12.4.4, "Semantics"*.

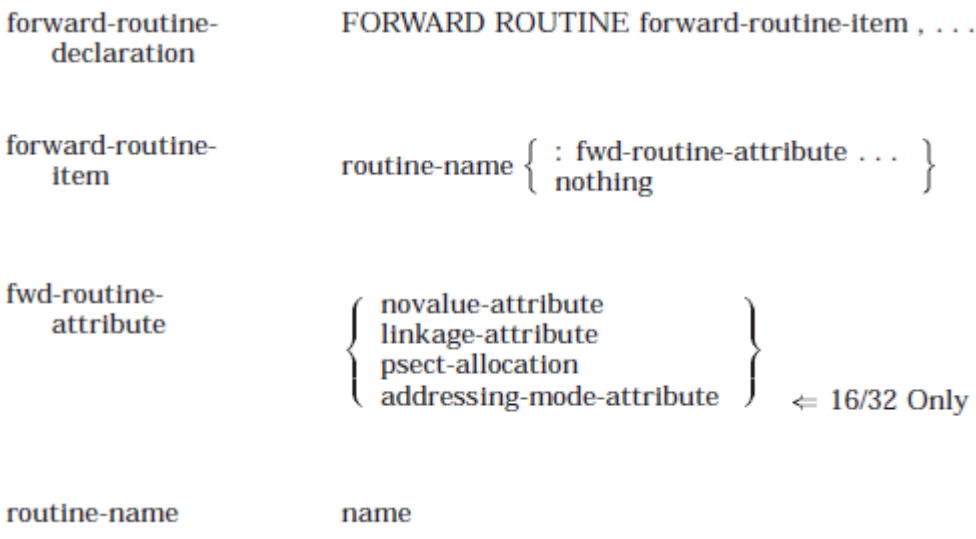
12.6. Forward-Routine-Declarations

Every routine must be declared by an ordinary- or global-routine declaration. Sometimes, however, it is necessary to use the routine-name before its full definition is given. Before you use the name, a forward-routine-declaration must be used to declare the name as a routine-name and to associate a limited set of attributes with it.

As an example of the use of a forward-routine-declaration, consider the two routines A and B. Routine A calls routine B and routine B calls routine A. If the ordinary-routine-declaration for A is given first, a forward-routine-declaration must be given for B. If the ordinary routine-declaration for B is given first, a forward-routine-declaration must be given for A.

In general, the use of a forward-routine declaration (at the beginning of a block) to specify all of the routine-names that are declared in the remainder of the block serves as a useful "table of contents" and allows the routines to be written in an order that is independent of their calling relationships.

12.6.1. Syntax



12.6.2. Restrictions

A routine-name declared in a forward-routine-declaration must appear in an ordinary- or global-routine-declaration later in the same block.

After any default attributes are filled in, a forward-routine-declaration must agree with its corresponding ordinary- or global-routine-declaration with respect to the set of attributes allowed in both declarations.

12.6.3. Semantics

A forward-routine-declaration declares a name to be a routine-name whose definition is given later in the same block, and associates with that name the set of attributes needed for generation of calls to the named routine. The semantics of the BLISS–32 addressing-mode-attribute (which is not one of the ordinary or global routine-attributes) is described in *Section 9.13, "The Addressing-Mode-Attribute – BLISS–16/32 Only"*.

12.7. External-Routine-Declarations

Often a routine must be defined in one block of a program and called in other blocks of the same program. Usually this situation arises from the organization of the program into separately compiled modules, but this need not be the case.

In order to provide for the linkage between routine-calls and routine definitions that occur in different scopes (for example, different modules), external-routine-declarations must be used. Specifically, the routine-name is declared in one block by a global-declaration (which defines the routine) and is declared in the other blocks by external-declarations.

12.7.1. Syntax

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

external-routine-item           routine-name { : ext-routine-attribute . . . }
                                             { nothing                       }

routine-name                    name

ext-routine-attribute           { novalue-attribute                 }
                                { linkage-attribute                 }
                                { psect-allocation                  }
                                { addressing-mode-attribute  <= 16/32 Only
                                { weak-attribute             <= 32 Only
```

12.7.2. Restrictions

A name must not be declared an external routine unless it is declared a global routine or a global bind routine in some other block of the same program. This restriction does not apply, however, to an external

name that is declared with the weak-attribute (BLISS–32 only; see *Section 9.14, "The Weak-Attribute – BLISS–32 Only"*).

12.7.3. Semantics

An external-routine-declaration informs the compiler that the definition of the routine-name is not in the current block. The compiler takes note of the attributes given in the external-routine-declaration. Then, each time a use of the declared routine-name is encountered, the compiler leaves a blank space in the object code for the routine-address. Later, the linker fills in the blank with a specific address.

The attributes in an external-routine-declaration provide the information the compiler and linker need to proceed in the absence of a full routine-declaration in the same module. The linkage attribute gives the compiler information about the type of call to generate for the routine and the availability and uses of registers within the routine. In particular, the novalue-attribute permits the compiler to detect an invalid call on the routine (a call that expects a value). The addressing-mode-attribute and weak-attribute (BLISS–32 only) are described in *Chapter 9, "Attributes"*.

Chapter 13. Linkages

A *linkage* is the particular calling-sequence convention used in calling a routine, and the register-management discipline used during execution of the routine that is called. The type of object code generated by the compiler for a routine-call is determined by the *linkage-definition* associated with the called routine. The linkage-definition also controls the object code generated for the entry and exit sequences of the routine with which it is associated. Thus, a linkage serves as the bridge between a routine and any routines that call it.

A linkage-definition may be explicitly declared in a *linkage-declaration*. Each BLISS dialect also provides several predefined linkages: one designed for standardized calls between BLISS–compiled routines (used as the default linkage), and others for calls between BLISS–compiled routines and FORTRAN-compiled routines. In the case of BLISS–36, a predefined linkage is also provided for compatibility with BLISS–10.

Each linkage-definition, whether predefined or explicitly declared, is identified by a *linkage-name*. Every routine, in turn, has a linkage-name associated with it, either by default or by explicit specification of a linkage-attribute in the routine's declaration.

The BLISS linkage facility consists of the following features:

- Linkage-declarations
- Predeclared linkage-names
- Linkage-functions (a class of executable-functions)
- Global-register-declarations
- External-register-declarations

This chapter describes the first three language features, and then discusses their use in conjunction with the global- and external-register-declarations. Primary descriptions of the register declarations are given in *Chapter 10, "Data Declarations"*.

In general, the BLISS linkage facility provides a type of control over the compiled code that is quite unusual in high-level languages, but which is often needed for efficiency-sensitive system applications. It allows, when necessary, a high degree of control over the kind of calling sequence generated by the compiler, and the register-usage conventions that are observed by related routines.

This control might be exercised, for example, in order to optimize a given routine or group of routines (for example, a subsystem) in terms of either size or execution time, or to produce a BLISS routine suitable for use with software written in other languages.

13.1. Introduction to Linkage-Declarations

A linkage-declaration declares a linkage-name that is defined by a particular combination of linkage characteristics. These characteristics include:

- Linkage-type – The general type of calling sequence, in terms of the specific transfer-of-control instructions and/or the software calling convention.
- Parameter-location options – The method by which actual-parameters are passed.

- Register-usage options – Specification of the registers that are saved and restored across a call, and of those that will not be used in a called routine.
- Global-register options – Specification of register data segments that are shared between routines.

The linkage-declarations of each BLISS dialect are quite system-specific; they are tailored to the particular hardware capabilities of each system and to the major software calling conventions in use on those systems. Nonetheless, there are many aspects of linkage-declarations that apply to two or more of the BLISS dialects.

This introduction to linkage-declarations explains the common aspects in three sections. The first discusses the many ways that registers can be used. This section is especially important because it establishes much of the vocabulary and many of the concepts used throughout this chapter. The second section presents a partial syntax for linkage-declarations that includes constructs common to at least two of the BLISS dialects. The third section describes the parts of the linkage-declaration and further develops the concepts introduced in the first section.

13.1.1. Register Usage

During the execution of a routine, some temporary storage is usually needed for holding values until they are used. The stack frame associated with the execution of the routine is one place to hold such values and the general registers are another. The general registers are more often preferable to the stack frame because they can be accessed more quickly and/or with shorter instructions. However, when one routine calls another, some consistent rules regarding register usage must be observed in order for both to use the machine registers correctly. The different uses of these registers can be broadly classified as special purpose and general purpose. *Special purpose registers* are dedicated for the same particular purpose among a group of routines; frequently that group is all of the routines of a program. *General purpose registers* are used for a variety of purposes by different routines and even within a single routine. This classification is hardly precise and does not even consider certain other kinds of usage that are described later; but it does provide a basis for discussion.

13.1.1.1. Special Purposes

In BLISS there are five types of special purposes to consider for register usage: program counter, stack pointer, frame pointer, argument pointer, and value-return register. As will be seen, registers are not dedicated for all of these purposes in every routine.

The *program counter register* is used to contain the address of the next instruction to be executed. In BLISS–16, the program counter is always register 7, and in BLISS–32 it is always register 15. In BLISS–36, the program counter is a special, not generally accessible part of the machine architecture, and thus does not figure in BLISS–36 register assignments.

The *stack pointer register* is used to contain the address of a portion of memory used for temporary storage during the execution of each routine. When a routine is called, the stack pointer is adjusted to point to a new area and when the routine returns the previous address is put back. The stack pointer may be adjusted many times during the execution of the routine as the need for temporary storage grows and diminishes in different parts of the routine. The portion of storage between the original address in the stack pointer and the current value at any particular point in time is known as the *stack frame* for that call of the routine.

Stack frames can vary greatly in size and complexity. A stack frame might be as small as a single fullword containing the program counter for returning to the calling routine or it might be very large, containing many values, fields, addresses, preserved register values, and so on.

The *frame pointer register* is used to contain the address of a fixed part of the stack frame of a routine. In contrast with the stack pointer, which may be adjusted many times during the execution of a routine, the frame pointer is generally set once at the beginning of routine execution and only changes when another routine is called and when the routine completes and returns. The utility of a frame pointer comes from this stable characteristic; the frame pointer makes access to fixed parts of the stack frame simple and efficient.

The *argument pointer register* is used to contain the address of a block of storage that contains the values of the actual-parameters of a routine-call.

The *value return register* is a register used to contain the value of a routine during the process of completion and returning.

The value return register, unlike the other special registers, is used as such only briefly during the completion of one routine and the resumption of the calling routine. Consequently, this register can also be used for general purposes during the execution of a routine.

13.1.1.2. General Purposes

A register that is not dedicated to one of the special purposes described in the preceding section can be used in a variety of ways. These uses are divided as follows:

- Locally usable (Preserved)
- Locally usable (Nonpreserved)
- Globally usable
- Not used

A *preserved register* contains the same value after returning from a routine-call as it contained at the time the routine was called.

A *nonpreserved register* does not (necessarily) contain the same value after returning from a routine-call as it contained at the time the routine was called.

Preserved and nonpreserved registers are together called *locally usable registers*. This combined designation is convenient because many of the rules concerning register usage apply equally to both preserved and nonpreserved registers.

Locally usable registers are used by the compiler according to its optimization strategies. The compiler determines how many of them to use, which to use for evaluating expressions, which to allocate for local data segments, and so on.

A *globally usable register* is used to contain a global register data segment, that is, a register data segment that is accessible in more than one routine. Global register data segments are governed by special rules involving LINKAGE declarations in combination with GLOBAL REGISTER and EXTERNAL REGISTER declarations. See *Section 13.7, "Global Register Data Segments and Linkages"* for complete details.

A *not used register* is simply not used in any way (applicable to BLISS–32 only).

13.1.1.3. Other Purposes

Registers can also be used to pass the values of actual-parameters of a routine-call to the routine that is called. These registers must be among the locally usable registers of the called routine. When such

an actual-parameter is evaluated, the value is assigned to a given register instead of to a position in an argument block or the stack. The routine that is called can efficiently fetch such a parameter value because it is already available in a register at the beginning of the routine execution.

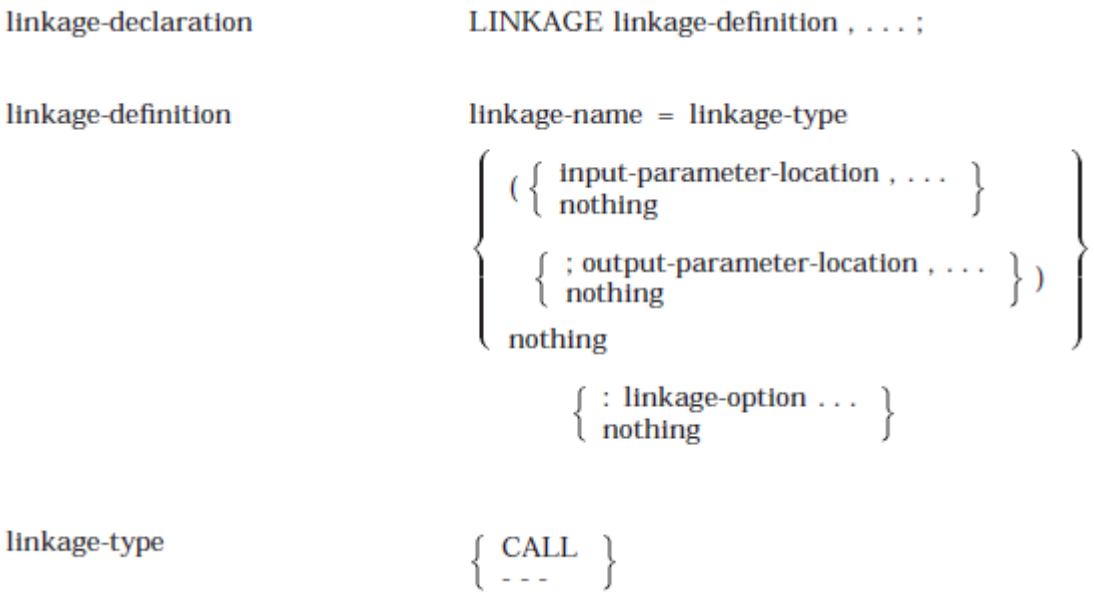
One or more of the locally usable registers can be allocated for a data segment established by a REGISTER declaration (see *Section 10.7, "Register-Declarations"*).

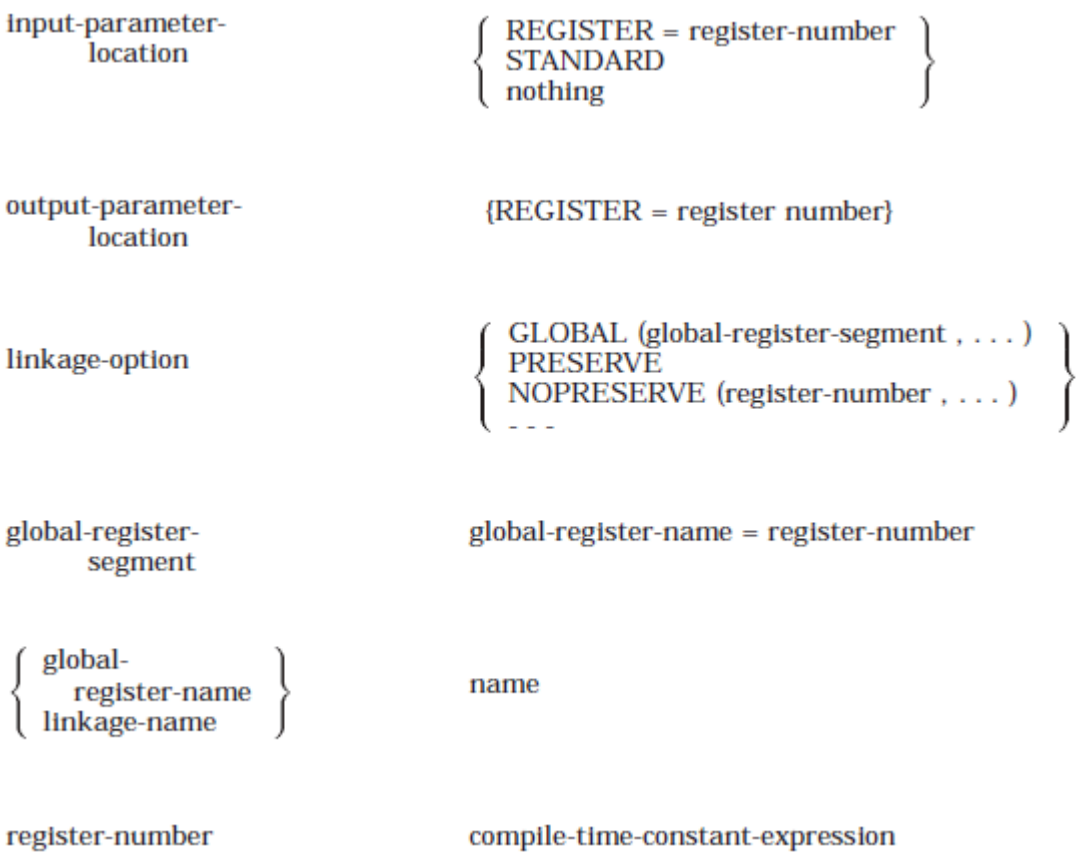
13.1.1.4. Multiple Purposes

Most registers are not limited to a single purpose or class of purpose. The program counter and stack pointer in both BLISS–16 and BLISS–32, as well as the frame pointer in BLISS–32, are truly dedicated by the hardware for these purposes; but these are the only cases.

Registers can be used for multiple purposes so long as those uses do not conflict. Because of the many different kinds of use, the rules for compatible use are complicated and lengthy. Even so, BLISS still does not always allow every imaginable combination; that would get even more complicated and lengthy. But, by and large, BLISS does allow nearly all of the register uses and combinations of uses that play a significant role in system software on each of the target systems.

13.1.2. Typical Syntax





The notation "- - -" in the above diagram indicates that there are additional alternatives in some of the dialects that are not shown. This syntax diagram does not apply completely to all of the BLISS dialects, but it is representative. For example, the CALL linkage-type is part of BLISS–16 and BLISS–32, but not BLISS–36.

13.1.3. Restrictions

In BLISS–16, the CALL linkage type is valid with input-parameter-locations, but not with output-parameter-locations.

The general-registers referenced by output-parameter-locations are implicitly NOPRESERVE, and cannot appear in NOTUSED, PRESERVE, or GLOBAL linkage modifiers; however, they may appear in NOPRESERVE modifiers, but this is not required.

A register-number value must not be given as both a parameter-location and a global-register-segment, and must not be given in more than one parameter-location or global-register-segment,

A register-number value must not be given in more than one linkage option.

13.1.4. Semantics

The same register may be both an input- and an output-parameter-location.

Each output-parameter-location specifies that a result from the evaluation of the routine-body will be returned in that register.

The output-actual expressions in the routine-call are associated with the output-parameter-location registers specified by the linkage-declaration. When the routine returns to the caller, the contents of each output-parameter-location register is assigned to the output-actual field reference.

If fewer output-actual expressions are present than were specified by the linkage, the remaining output-parameter-location registers are treated as NOPRESERVE's. If an empty element (identified by a null expression) appears in the list, it will (when output-actuals are bound to the appropriate output-parameter-location registers) be treated as a placeholder.

The linkage-declaration defines a name for a particular combination of calling sequence characteristics. A name so declared can be used as a linkage-attribute in any kind of routine-declaration. The several parts of a linkage-definition are described in the following sections.

13.1.4.1. Linkage-Types

The linkage-type selects the principal characteristics of the calling sequence to be used. Each linkage-type generally establishes the following:

- The specific machine instructions to be used to transfer control to a routine and to return from the routine.
- Whether or not an argument pointer is used to address actual-parameter values.
- Which linkage-options are applicable.
- The defaults for linkage-options.

The CALL keyword occurs as a linkage-type in BLISS–16 and BLISS–32; however, the only common characteristic that CALL implies is the use of an argument pointer to access actual-parameters (that is, input- and output-actuals for BLISS–32, and input-actuals only for BLISS–16). CALL is not the only linkage-type that implies use of an argument pointer; the F10 linkage-type in BLISS–36 also implies use of an argument pointer.

13.1.4.2. Parameter-Locations

An input-actual-parameter of a routine-call can be passed to the called routine in one of two ways: it can be passed in a standard, or default, method or it can be assigned to one of the general registers; however, an output-actual-parameter must be assigned to one of the general registers.

There are two major variations on the standard method; the linkage-type determines which one is used. The two methods are:

- By *argument pointer*
- By *implicit stack location*

Argument Pointer Method

In the argument pointer method, all of the input-actual-parameters of the routine-call are assigned to successive positions in a block called the *argument block*. The address of this block is passed to the called routine using one of the general registers. A register used in this way is called an *argument pointer register*. The called routine fetches an input-actual-parameter value from the argument block, using the argument pointer value in combination with an offset determined from the formal-name that corresponds to that input-actual-parameter position.

In addition to the input-actual-parameter values, an argument block can contain additional information concerning the parameter values. In each BLISS dialect, the argument block contains the number of input-actual-parameter values in the block. In BLISS–36 other information may also be contained in the argument block.

An argument block may be located anywhere in storage at the option of the compiler. It might be part of the stack frame of the routine containing the routine-call or it might be in permanently allocated storage. A restriction against assigning to a formal-name assures that an argument block can be allocated in storage protected against writing and/or reused in the calling routine for other routine-calls.

Implicit Stack Location Method

In the implicit stack location method, the input-actual-parameters of the routine-call are assigned to successive positions in the stack frame of the routine containing the call. No explicit value giving the location of the parameters is passed to the routine that is called. The called routine fetches an input-actual-parameter value using implicit information about where the value is located in the stack frame.

Register Parameters

In addition to the standard method of passing input-actual-parameter values, some or all of the parameters can be passed by assigning them to specified general registers. This method can be used in combination with the standard method; for example, one parameter can be passed in a register, and the others in the standard way. However, all output-actual-parameters must be passed by the general-register method.

The general-registers referenced by output-parameter-locations are implicitly NOPRESERVE, and cannot appear in NOTUSED, PRESERVE, or GLOBAL linkage modifiers. The registers may appear in a NOPRESERVE linkage-option, but such specification is unnecessary.

13.1.5. Linkage-Options

Linkage-options supplement and modify the basic calling sequence conventions established by the linkage-type. For example, in BLISS–36 the LINKAGE_REGS option can be used in combination with the PUSHJ linkage-type to specify the registers to be used as the stack pointer, frame pointer, and value-return register, respectively, if the default choices for the PUSHJ linkage-type are not suitable.

In some cases, a particular linkage-option must only be used in combination with a specific linkage-type. The LINKAGE_REGS option just mentioned is an example; it must only be used with the PUSHJ linkage-type in BLISS–36.

In a few cases, linkage-options can be used with several linkage-types and in more than one BLISS dialect. The PRESERVE, NOPRESERVE, and GLOBAL linkage-options are examples. They can be used in all dialects with at least two different linkage-types.

In the object code generated for a given routine, each register's use is governed by one of three usage conventions, each corresponding to one of the following linkage-option keywords:

PRESERVE

A *preserved register* can be used during the execution of the routine, but the original contents at the time of the routine call must be restored at the time the routine completes and returns.

NOPRESERVE

A *nonpreserved register* can be used during the execution of the routine (without restoring its original contents).

GLOBAL

A *globally usable register* is used only as determined by its corresponding GLOBAL REGISTER and EXTERNAL REGISTER declarations, and by explicit source-code references to such a register.

A register that is given in a PRESERVE linkage-option contains the same value after returning from a routine as it contained at the time the routine was called. The called routine may or may not use the register. If it does, then special action is taken to save the contents of the register (push it onto the stack) before the register is used and restore it (pop it from the stack) afterward. If the register is not used, then no special action is needed. In either case, a calling routine is able to leave useful information in a register preserved by the routine being called – the information is still available after the call.

A register that is given in a NOPRESERVE linkage-option does not necessarily contain the same value after returning from a routine as it contained at the time the routine was called. The called routine may or may not use the register, but in either case no special action is taken to preserve its contents. A calling routine must not leave needed information in a register that is not preserved by the routine being called – the information may not be available after the call.

Registers that are given in a GLOBAL linkage-option are used to contain global register data segments by both calling and called routines. Globally usable registers are not managed by the compiler; they are used only as explicitly directed by the source program. In certain special cases, depending on the linkage-type and other details, a register given in a GLOBAL linkage-option may be treated as a preserved register, rather than as globally usable. These cases are described later in the sections for each BLISS dialect.

Globally usable registers are described fully in *Section 13.7, "Global Register Data Segments and Linkages"* where the GLOBAL linkage-option and the related GLOBAL REGISTER and EXTERNAL REGISTER declarations are considered together.

13.2. BLISS–16 Linkage-Declarations

The linkage capabilities provided by the linkage-declaration in BLISS–16 are the following:

- The JSR, CALL, EMT, TRAP, IOT, INTERRUPT, and RSX_AST linkage-types
- Standard or register parameter-locations for input-actuals and register parameter-locations for output-actuals
- Globally used and locally used registers
- The CLEARSTACK, RTT, and VALUECBIT exit sequence linkage-options

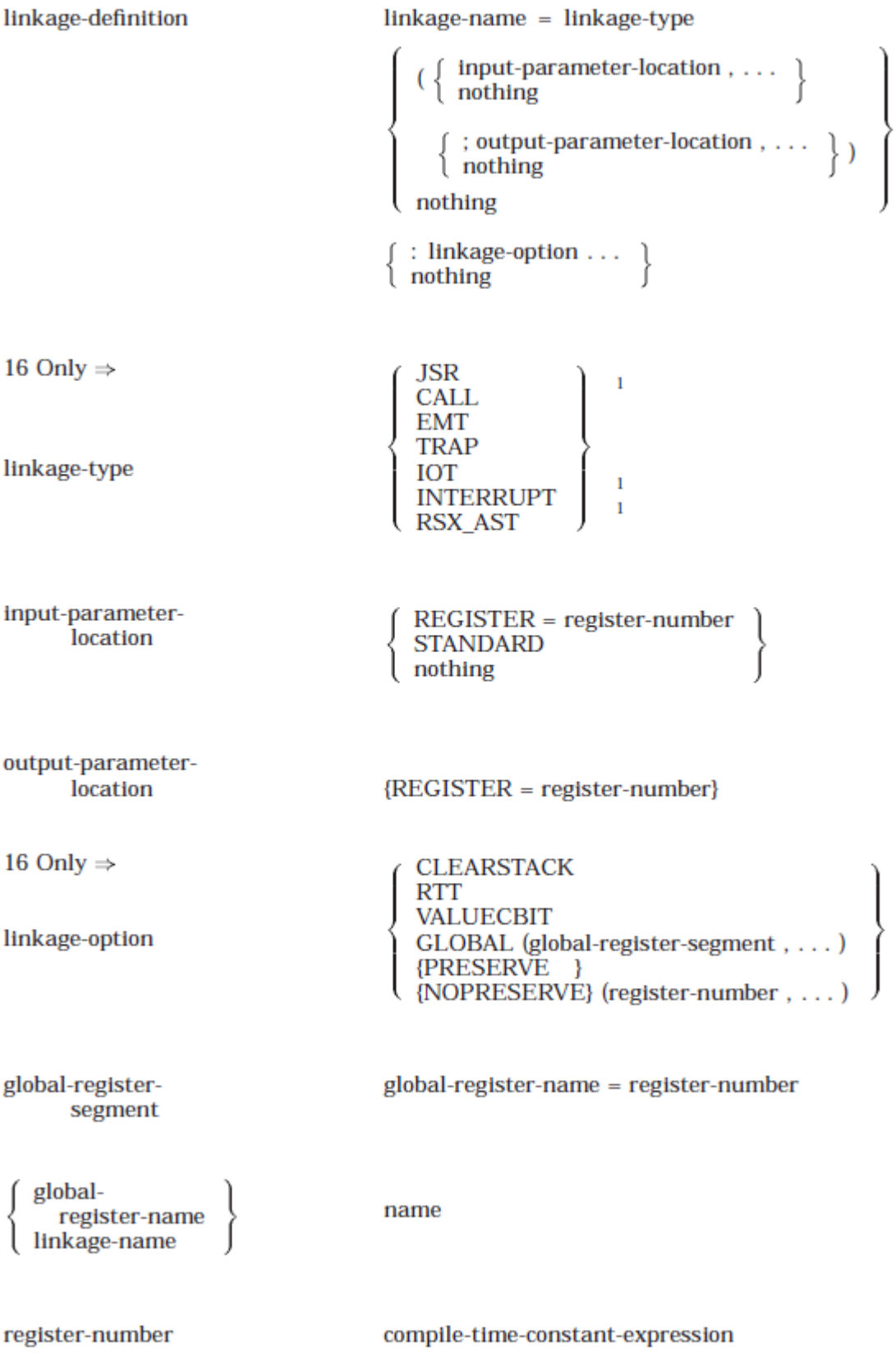
As an example of a linkage-declaration, consider the following:

```
LINKAGE
   PAR2REG3 = CALL(STANDARD, REGISTER = 3);
```

The declaration indicates that the CALL linkage-type is to be used and that the second input-actual-parameter is to be passed using register 3. The first input-actual-parameter and any parameters after the second parameter are to be passed in the standard way.

13.2.1. Syntax

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



1. Linkage-type is invalid with output-parameter-locations.

13.2.2. Restrictions

Linkage-names defined with EMT, TRAP, or IOT linkage-types can be used only as a linkage-attribute in bind, global bind, and external routine declarations (or in a general-routine-call as described in *Section 13.2.4.2, "EMT, TRAP, and IOT Linkage-Types"*).

The register-number value must be in the range 0 to 5.

A register-number value must not be given as both a parameter-location and a global-register-segment, and must not be given in more than one parameter-location or global-register-segment.

A register-number value must not be given in more than one linkage-option.

If the CALL linkage-type is given, then the register-number of a REGISTER parameter-location must be in the range 0 to 4.

The GLOBAL, PRESERVE, NOPRESERVE, CLEARSTACK, and VALUECBIT linkage-options must not be specified with the CALL linkage-type.

If OTS (run-time library) routines are called, register 0 must not be specified as a global-register-segment in the calling routine's linkage-definition.

If the CLEARSTACK linkage-option is given, the number of actual-parameters in a (general) routine-call must be equal to the number of parameter-locations given.

The VALUECBIT linkage-option may not be specified in a linkage-definition for a routine written in BLISS.

If the VALUECBIT linkage-option is given, the CLEARSTACK linkage-option must also be given.

The RTT linkage-option must be given only with the INTERRUPT linkage-type. No linkage-option can be given with the RSX_AST linkage-type.

13.2.3. Defaults

If a parameter-location is not given, then STANDARD is assumed. If a routine-call or routine-declaration contains more parameters than are given in the associated linkage-definition, then STANDARD is assumed as the parameter-location for each of the additional parameters.

For the JSR linkage-type, the registers are used as follows, by default:

Registers	**Default Usage**
0	Value return register, nonpreserved
1–5	Preserved
6	Stack pointer
7	Program counter

For the CALL linkage-type, the registers are used as follows:

Registers	**Usage**
0	Value return register, nonpreserved
1–4	Preserved

5	Argument pointer
6	Stack pointer
7	Program counter

The default usage cannot be modified for the CALL linkage-type.

For the EMT, TRAP, IOT, INTERRUPT, and RSX_AST linkage-types, the registers are used as follows, by default:

Registers	**Default Usage**
0–5	Preserved
6	Stack pointer
7	Program counter

13.2.4. Semantics

A linkage-definition defines a name that designates a particular combination of calling sequence options. Generally, such a name can be used as a linkage-attribute in any kind of routine-declaration; however, this is not true of all linkage-names.

The linkage-type JSR specifies that the PDP–11 JSR and RTS instructions are used by the compiled code, and that the parameters with STANDARD parameter-locations are placed on the stack (without a parameter count) and accessed by the called routine relative to the stack pointer (SP) register.

The linkage-type CALL specifies that the PDP–11 JSR and RTS instructions are used by the compiled code, and that the parameters with STANDARD parameter-locations are passed using register 5 (R5) as the argument pointer.

The linkage-types INTERRUPT and RSX_AST specify that a routine will be called only by a PDP–11 hardware or software interrupt. These linkages are further described in *Section 13.2.4.1, "INTERRUPT Linkage-Type"* and *Section 13.2.4.3, "RSX_AST Linkage-Type"*.

If REGISTER is specified for a parameter-location, the given register will be used as the location to which the actual-parameter value is to be assigned, and correspondingly, is the location where the called routine expects to find the value. This use of a register location to transmit an actual-parameter value to a called routine does not affect the semantics associated with the use of the corresponding formal-parameter name.

The CLEARSTACK linkage-option (which can be used only with the JSR, EMT, TRAP, IOT, or INTERRUPT linkage-type) specifies that the actual-parameters that are placed on the stack for a routine-call are removed from the stack by the called routine (instead of by the calling routine). If CLEARSTACK is not specified, they will not be removed by the called routine (and are the responsibility of the caller). The VALUECBIT linkage-option (which can be used only with the JSR, EMT, TRAP, IOT, or INTERRUPT linkage-type, and only in combination with CLEARSTACK) specifies that an external routine declared with this linkage-option returns its value in the C bit, and that the value of register 0 is undefined on return from such a routine. This linkage-option is used to communicate with non-BLISS routines having this value-return characteristic.

The RTT linkage-option (which can be used only with the INTERRUPT linkage-type) specifies that the PDP–11 RTT instruction should be used to exit from the interrupt routine instead of the normal RTI instruction.

The GLOBAL, PRESERVE, and NOPRESERVE linkage-options specify the usage conventions that apply to each PDP–11 machine register at the time a routine is called and during the execution of the routine. There are three conventions, one corresponding to each of the three linkage-option keywords. You specify a usage convention for a register by giving its number in the appropriate linkage-option. The description of these linkage-options is given in *Section 13.1.5, "Linkage-Options"*.

Register usage conventions can be specified only for registers 0 through 5; the remaining registers (the stack pointer and program counter) are used only as specified in the PDP–11 hardware and software architecture.

Globally usable registers are not managed by the compiler; they are used only as explicitly given in the source program.

13.2.4.1. INTERRUPT Linkage-Type

A linkage-name defined with the INTERRUPT linkage-type can be used only as a linkage-attribute in a forward-, ordinary-, or global-routine declaration. It specifies that the routine to which it is applied will be invoked only by a PDP–11 hardware interrupt or software simulation of an interrupt (such as an RSX–11 Synchronous System Trap). Interrupts may occur as a result of certain external events, such as I/O device completion, or as a result of programmed events, such as execution of certain instructions: EMT, IOT, and so on. See *Section 13.2.4.3, "RSX_AST Linkage-Type"* concerning the related linkage-type RSX_AST.

The number of formal-names given for the routine must equal the number of values pushed on the stack by the call. In most cases this is exactly two. However, interrupt routines that are called by general-routine-calls using a linkage-name defined with an EMT, TRAP, or IOT linkage-type can have more than two formal parameters.

The formal parameters of the routine correspond to the hardware values in the order pushed; that is, the first formal parameter corresponds to the first value pushed, the second formal parameter corresponds to the second value pushed, and so on. Consequently, the first formal parameter corresponds to the pushed processor status (PS) and the second formal parameter corresponds to the pushed program counter (PC).

13.2.4.2. EMT, TRAP, and IOT Linkage-Types

In a general-routine-call that uses a linkage-name defined with an EMT, TRAP, or IOT linkage-type, the following special rules apply:

- For EMT and TRAP, the first value in the actual-parameter list is not interpreted as a routine-address. Instead it is interpreted as a value that is incorporated into the low byte of the EMT or TRAP instruction itself. It must be a compile-time constant expression in the range 0 to 255.
- For IOT, all of the values in the parameter list are actual-parameters. There is no routine-address parameter.

13.2.4.3. RSX_AST Linkage-Type

Similar to the INTERRUPT linkage-type, the RSX_AST linkage-type specifies that the routine to which it is applied will be invoked only by an RSX–11 Asynchronous System Trap (AST). The first four formal parameters of such a routine are mandatory and correspond to the following context information:

1. Event-flag mask word
2. Program status word
3. Program counter

4. Directive Status Word of the interrupted task

Additional formal parameters must be specified if the kind of AST that invokes the routine pushes supplemental information onto the stack. At the routine's return point, any such supplemental information is removed from the stack and an RSX–11 AST SERVICE EXIT directive (rather than an RTS instruction) is executed.

13.2.5. BLISS–16 Predeclared Linkage-Names

Four linkage-names are predeclared in every BLISS–16 module. The linkages are provided for compatible and transportable usage among the several BLISS dialects. See *Section 13.5, "Common Predeclared Linkage-Names"* concerning such usage.

The predeclared linkage-names are defined as shown in the following declaration:

```
LINKAGE
   BLISS = JSR,
   FORTRAN = CALL,
   FORTRAN_SUB = CALL,
   FORTRAN_FUNC = CALL;
```

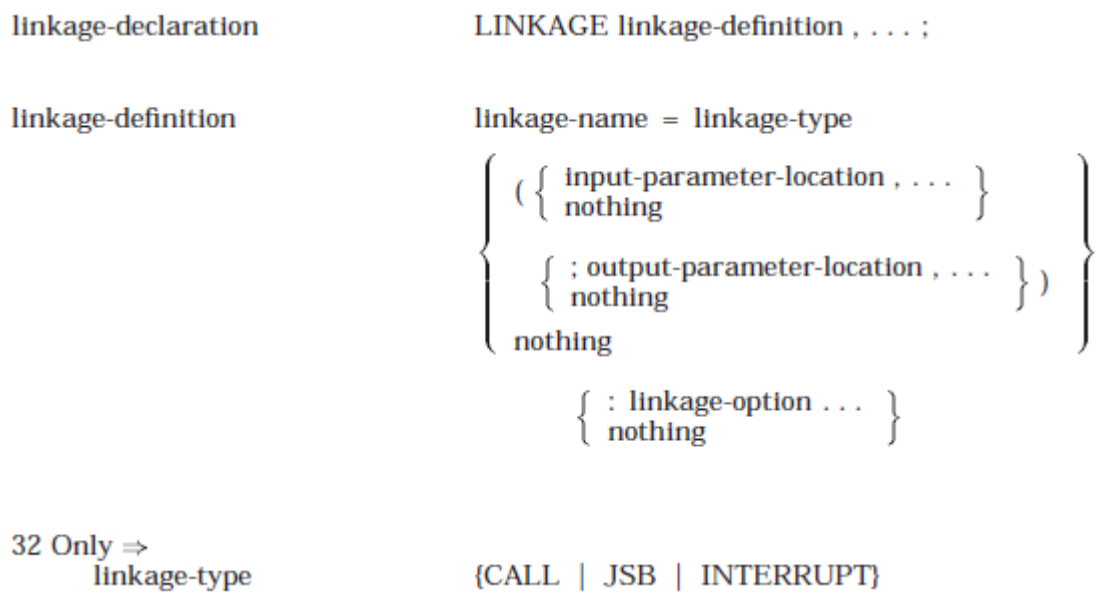
13.3. BLISS–32 Linkage-Declarations

A linkage-declaration in BLISS–32 can be used to specify a CALL, JSB, or INTERRUPT linkage-type, to designate registers for passing parameters, and to identify registers as globally used, locally used, or not used. For example:

```
LINKAGE
    DBL_PREC = CALL( ; REGISTER=0, REGISTER=1);
```

The declaration indicates that the CALL linkage-type is to be used and that output-actual-parameters are to be passed using registers 0 and 1 for a double-precision result. Because the registers are treated as output-parameter locations, the called routine (DBL_PREC) should be declared as NOVALUE.

13.3.1. Syntax



13.3.2. Restrictions

A NOTUSED linkage-option must only be given with the JSB and INTERRUPT linkage-types. It must not be given in combination with the CALL linkage-type.

The register-number in a REGISTER parameter-location or a linkage-option must be in the range 0 to 11.

A register-number value must not be given as both a parameter-location and a global-register-segment, must not be given as both a parameter-location and in a NOTUSED linkage-option, and must not be given in more than one parameter-location or global-register-segment.

A register-number value must not be given in more than one linkage-option.

Some of the character-handling and machine-specific functions require the use of particular machine registers because they result in VAX instructions that use specified registers; such functions must not be used if the required registers are not locally usable. Observe that at most the set of registers 0 through 5 inclusive must be locally usable to satisfy this requirement.

The VAX calling standard requires that register 0 or registers 0 and 1 together be used to return routine values. This requirement, combined with the preceding general restriction, leads to the following two special-case restrictions:

- If a routine-call is in the scope of a global register data segment that is allocated in either register 0 or 1, then the routine that is called must not return a value; that is, must be declared with the NOVALUE attribute.

- If the linkage-attribute of a routine-declaration specifies registers 0 or 1 as PRESERVE, GLOBAL, or NOTUSED, then that routine must also have the NOVALUE attribute.

The VAX calling standard also requires that registers 0 and 1 be usable as temporary registers by the condition handling software during processing of a signal (see *Chapter 17, "Condition Handling"*). Further, only routine stack frames associated with the CALL linkage-type are used for restoring register contents during unwinding. These requirements, together with the above restrictions on linkages, lead to the following special-case restrictions:

- A routine-body must not immediately contain an ENABLE declaration if the linkage-attribute of the routine is defined with linkage-type JSB, or INTERRUPT, or with registers 0 or 1 as either PRESERVE, GLOBAL, or NOTUSED.

- A routine whose linkage-attribute is defined with registers 0 or 1 as PRESERVE, GLOBAL, or NOTUSED must not be terminated by unwinding.

- If a routine-call to a routine with JSB linkage-type occurs in a routine with JSB linkage-type, all of the locally usable registers of the called routine must also be given as locally usable registers of the routine containing the call. That is, the outermost JSB routine in a nest of JSB routines must specify all the registers that are locally usable. This restriction assures that the CALL routine that calls the outermost JSB routine can preserve all the necessary registers.

The VAX calling standard is described in the *VAX Architecture Handbook*. Condition handling and its interaction with linkages are described in *Chapter 17, "Condition Handling"* of this manual.

13.3.3. Defaults

If a parameter-location is not given, then STANDARD is assumed. If a routine-call or routine-declaration contains more parameters than are given in the associated linkage-definition, then STANDARD is assumed as the parameter-location for each of the additional parameters.

For the CALL linkage-type, the registers are used as follows, by default:

Registers	**Default Usage**
0	Value return register, nonpreserved
1	Nonpreserved
2-11	Preserved
12	Augment pointer
13	Frame pointer
14	Stack pointer
15	Program counter

For the JSB linkage-type, the registers are used as follows, by default:

Registers	**Default Usage**
0	Value return register, nonpreserved
1	Nonpreserved

2-11	Preserved
12-13	Not used
14	Stack pointer
15	Program counter

Observe that, for both CALL and JSB linkage-types, registers 0 to 11 are locally usable by default.

For the INTERRUPT linkage-type, the registers are used as follows, by default:

Registers	**Default Usage**
0–13	Preserved
14	Stack pointer
15	Program counter

13.3.4. Semantics

A linkage-declaration defines a name for a particular combination of calling sequence options. A name so declared can be used as a linkage-attribute in any kind of routine-declaration.

The linkage-type CALL specifies that the VAX CALLS, CALLG, and RET instructions are used. Further, the parameters with STANDARD parameter-locations are passed using register 12 (AP) as the argument pointer.

The linkage-type JSB specifies that the VAX JSB, BSBW, BSBB, and RSB instructions are used by the compiled code. Further, the parameters with STANDARD parameter-locations are placed on the stack (without a count) and accessed by the called routine relative to the stack pointer (SP) register. If REGISTER is given as a parameter-location, then the given register is used as the location to which the actual-parameter value is assigned in performing a routine-call, and correspondingly, is the location where the called routine expects to find the actual-parameter value. This use of a register location to transmit an actual-parameter value to a called routine does not affect the semantics associated with the use of the corresponding formal-parameter name.

The linkage-options specify the usage conventions that apply to each VAX machine register at the time a routine is called and during the execution of the routine. There are four conventions, one corresponding to each of the four linkage-option keywords, GLOBAL, PRESERVE, NOPRESERVE, and NOTUSED. A usage convention is specified for a register by giving its number in the appropriate linkage-option. The description of these linkage-options is given in *Section 13.1.5, "Linkage-Options"*.

Register usage conventions can be specified only for registers 0 through 11; the remaining registers (the argument pointer, frame pointer, stack pointer, and program counter) are used only as specified in the VAX hardware and software architecture.

Globally usable registers are not managed by the compiler; they are used only as explicitly given in the source program, with the following exception:

In a routine with a linkage that specifies CALL linkage-type and a globally-usable register (in a GLOBAL linkage-option), if the global-register-segment is not declared as a global register data segment (using an EXTERNAL REGISTER declaration) within the body of the routine, then the compiler can choose to consider the register preserved (and hence, locally usable).

However, in a routine with a linkage that specifies JSB linkage-type, the compiler cannot preserve and use such registers. The reason for the difference has to do with the requirements for condition handling. Briefly, the CALL linkage-type provides the information needed for the condition handling software to properly recover register values when doing unwinding; the JSB linkage-type does not.

Registers that are given in a NOTUSED linkage-option are not used in any way. Only routines with a linkage that specifies the JSB linkage-type can have registers that are not usable.

Some guidelines concerning the choice of registers to specify in a NOTUSED linkage-option are discussed in *Section 13.7.2, "Guidelines for BLISS–16"*.

13.3.4.1. JSB Linkage-Type

The routine EXCHANGE in *Section 12.4.5, "Pragmatics"* is an example of a routine that can be made significantly smaller and faster by the use of a linkage-declaration such as the following:

```
LINKAGE
FAST = JSB(REGISTER = 0, REGISTER = 1);
```

When the linkage-attribute FAST is given for the routine EXCHANGE, the JSB linkage-type is used instead of the CALL linkage-type and the parameters are passed in registers 0 and 1.

When a set of routines with JSB linkage-type call one another, make sure that the locally usable registers of the calling routine include all the locally usable registers of any routine that it calls. For example, consider the following linkage-declarations:

```
LINKAGE
    JSB_ALL = JSB,
    JSB_NO11 = JSB: NOTUSED(11);
```

The linkage JSB_ALL specifies a JSB linkage-type. Because no linkage-options are given, the locally usable registers are registers 0 to 11. The linkage JSB_ NO11 also specifies a JSB linkage-type. Because the linkage-option indicates that register 11 is not used, the locally usable registers are registers 0 to 10.

Suppose the following routines are declared:

```
FORWARD ROUTINE
    ALPHA: JSB_ALL,
    BETA: JSB_NO11;
```

Then routine ALPHA can legitimately call routine BETA. But routine BETA must not call routine ALPHA because the set of locally usable registers of ALPHA is not a subset of the locally usable registers of BETA.

13.3.4.2. INTERRUPT Linkage-Type

The INTERRUPT linkage-type for BLISS–32 is used for the same purposes and provides the same functionality as that described for BLISS–16, and is similar to the JSB linkage-type. When used in a routine-declaration, a linkage-name defined with the INTERRUPT linkage-type affects the following:

- All registers are preserved.
- As necessary, registers are explicitly saved with PUSHL or PUSHR instructions.
- All references to formal-parameters are via the stack pointer (SP).

- At routine exit, all but the last two arguments are removed from the stack; these are assumed to be a valid program counter (PC) and processor status longword (PSL).
- A Return from Exception or Interrupt (REI) instruction is executed.

Input- or output-parameter-location REGISTER assignments are not permitted with INTERRUPT linkages.

The correct number of formal-parameters must be declared with an INTERRUPT linkage routine to ensure that the compiler cleans the stack on exiting the routine; a routine with less than two parameters is invalid.

An INTERRUPT linkage routine is implicitly declared NOVALUE. An example of an INTERRUPT linkage routine in BLISS–32 follows:

```
LINKAGE
    ARITH_EXCP= INTERRUPT: NOTUSED(3,4,5,6,7,8,9,10,11);
ROUTINE ARITH_EXCP_HDLR(CODE,PC,PSL): ARITH_EXCP=
    BEGIN
    CASE .CODE FROM SRM$K_INT_OVF_T TO SRM$K_FLT_UND_F OF
    SET
    ...
    ...
    TES
    END
```

The code in the example is expanded as follows:

```
ARITH_TRAP_HDLR:
    PUSHL   R0
    CASEL   4(SP) ,#1,#9
    .WORD   ...
    ...
    MOVL   (SP)+,R0
    ADDL2  #4,SP
    REI
```

Notice in the first line of the expanded code that only one register (R0) is needed. In the second line the exception is dispatched via the exception code. The register is then restored (MOVL), and the trap code is eliminated (ADDL2) before a return (REI) is executed.

Explicit calls are also permitted to routines declared with interrupt linkage. The caller treats such a call as if it were declared with a JSB linkage attribute; an exception being that the parameters are automatically removed from the stack by the called routine and not the caller. The parameter order is such that the caller's PC is always the first formal-parameter and will not appear as an actual-parameter in the explicit routine-call.

If an interrupt linkage routine exists (for example, SETPSL), that is invoked with only the PC and PSL as actual-parameters, the routine can be explicitly called with the following BLISS expression:

```
SETPSL( .NEWPSL ):
```

13.3.5. BLISS–32 Predeclared Linkage-Names

Four linkage-names are predeclared in every BLISS–32 module. These linkages are provided for compatible and transportable usage among the several BLISS dialects. See *Section 13.5, "Common Predeclared Linkage-Names"* concerning such usage.

The predeclared linkage-names are defined as shown in the following declaration:

```
LINKAGE
    BLISS = CALL,
    FORTRAN = CALL,
    FORTRAN_SUB = CALL,
    FORTRAN_FUNC = CALL;
```

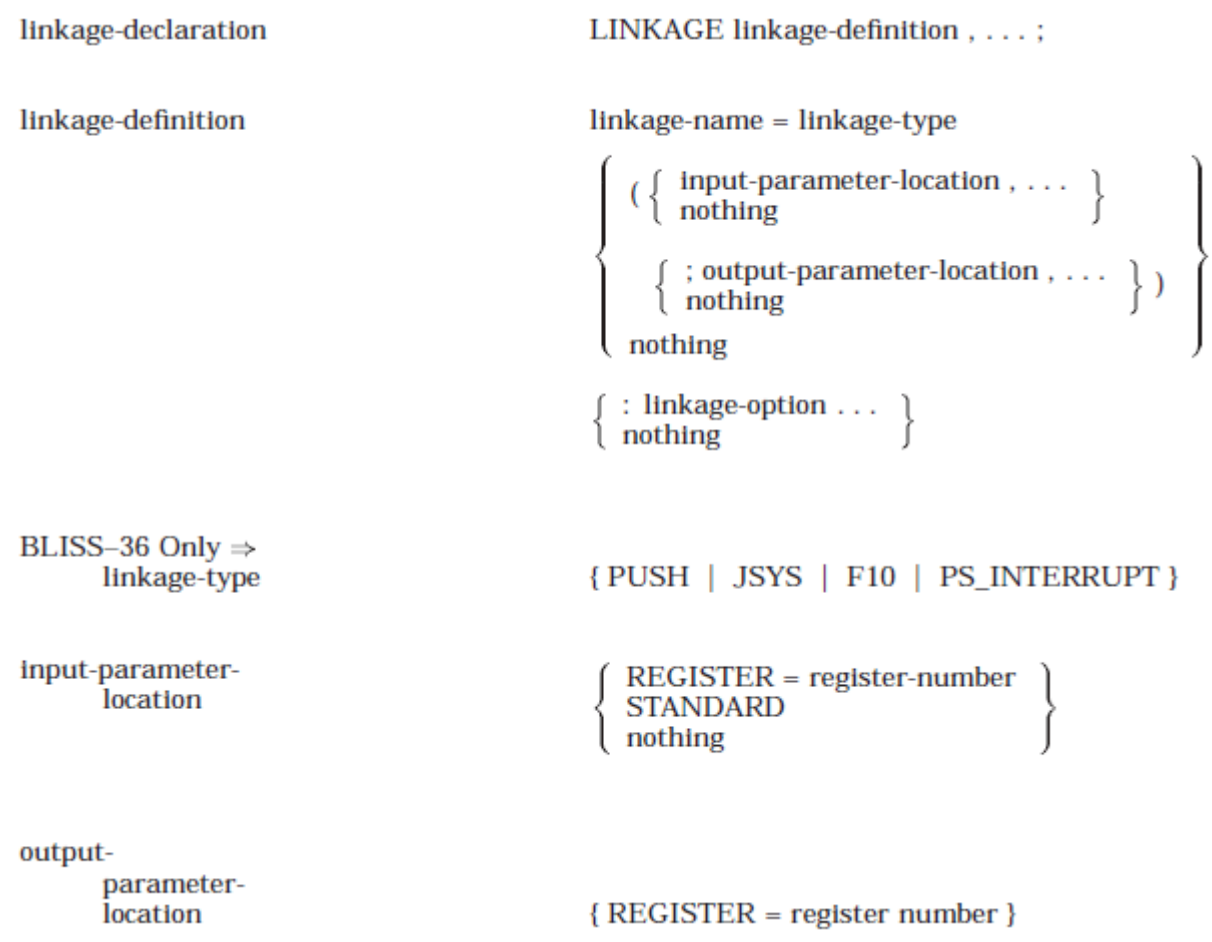
13.4. BLISS–36 Linkage-Declarations

A linkage-declaration in BLISS–36 can be used to specify a PUSHJ, JSYS, F10, or PS_INTERRUPT linkage-type, to identify globally used registers, to specify the use of a PORTAL instruction in the entry sequence of a routine, and to specify other linkage capabilities. For example:

```
LINKAGE
    PAR2REG4 = PUSHJ(STANDARD, REGISTER = 4);
```

The declaration indicates that the PUSHJ linkage-type is used and that the second actual-parameter is passed using register 4. The first actual-parameter and any parameters after the second parameter are passed in the standard way.

13.4.1. Syntax



BLISS–36 Only ⇒

linkage_option

{ general-linkage-option
pushj-linkage-option
ps_interrupt-linkage-option }

general-linkage-option

{ GLOBAL (global-register-segment , . . .)
PORTAL
{ PRESERVE
NOPRESERVE
SKIP(value)
CLEARSTACK } (register-number , . . .) }

pushj-linkage-option

LINKAGE_REGS (stack-pointer- reg,
frame-pointer-reg , return-value-reg)

ps_interrupt-linkage-option

{ PORTAL
LINKAGE_REGS (stack-pointer-reg,
frame-pointer-reg, return-value-reg }

{ stack-pointer-reg
frame-pointer-reg
return-value-reg }

register-number

global-register-segment

global-register-name = register -number

{ global-register-name
linkage-name }

name

register-number

compile-time-constant-expression

skip-value

-1 | 0 | 1 | 2

13.4.2. Restrictions

A REGISTER parameter-location (input or output) can be specified only with a PUSHJ or JSYS linkage-type.

Input- and output-parameter-locations cannot be specified with a PS_ INTERRUPT linkage-type.

The registers referenced by output-parameter-locations are implicitly NOPRESERVE and cannot appear in PRESERVE or GLOBAL linkage modifiers.

The register-number in a REGISTER parameter-location must be in the range 0 to 15 (JSYS excepted) and must not specify a register given as either the stack-pointer (SP) or the frame-pointer (FP). It can, however, be the same as the register given as the value-return register.

The register-numbers for the JSYS linkage must be in the range 1 to 4 (physical registers AC1 through AC4).

The LINKAGE_REGS linkage-option cannot be given in combination with a JSYS or F10 linkage-type.

Note

The JSYS built-in function is obsolete and should be avoided; instead, use the JSYS linkage.

When using a LINKAGE_REGS option with the PS_INTERRUPT linkage-type, you must supply all three register numbers, although the return-value-register is unused.

The SP register and the value-return register in the LINKAGE_REGS option must be in the range 0 to 15, and the FP register must be in the range 1 to 15. The register number in a linkage-option other than the LINKAGE-REGS option must be in the range 0 to 15 and must not specify a register used as a SP, FP, or argument pointer (if applicable).

All of the routines in a given program must use the same SP register, including any implicitly called Object Time System (OTS) routines. This restriction assures that a single OTS library can satisfy all of the requirements of a program.

The same register number value must not be given as both a parameter-location and a global-register-segment, and must not be given more than once as a parameter-location or a linkage-option register number. There is one exception: the register specified as the value return register in a LINKAGE_REGS option can also be specified as preserved, nonpreserved, or global.

If the value return register is also specified as preserved or global then the linkage-name so defined must only be used as a linkage-attribute in the declaration of a routine that also has the NOVALUE attribute or in a general-routine-call in a context that does not require a value.

The skip-values for the PUSHJ linkage-type are restricted to 0 through 2.

Some executable-functions impose hidden restrictions on the linkage-definition and explicit register usage of the containing routine. More specifically, some of the character-handling-functions and each of the condition-handling-functions result in calls to OTS routines.

These implicit routine calls are made with the governing OTS linkage for the program (BLISS36C by default). Therefore, any routine containing such functions must also be able to call a routine having the governing OTS linkage. In particular, the containing routine's use of register data segments declared by register-number, whether local or global, must be consistent with the register conventions of the OTS linkage. See the restrictions in *Section 10.7, "Register-Declarations"*, *Section 10.8, "Global-Register-Declarations"*, and *Section 10.9, "External-Register-Declarations"*.

13.4.3. Defaults

The defaults for each of the linkage-options depend on the linkage-type that is given.

Defaults for the PUSHJ Linkage-Type

If a parameter-location is not given, then STANDARD is assumed. If a routine-call or routine-declaration contains more parameters than are given in the associated linkage-definition, then STANDARD is assumed as the parameter-location for each of the additional parameters.

Default register usage for the PUSHJ linkage-type is determined in two steps: First, the defaults for the LINKAGE_REGS option are applied if the LINKAGE_ REGS option is not given; second, the defaults for all remaining registers are determined.

The default for the LINKAGE_REGS option is LINKAGE_REGS(0,2,3), which indicates the following:

Registers	**Default Usage**
0	Stack pointer
2	Frame pointer
3	Value return register, nonpreserved

For any register not specified by the explicit or default LINKAGE_REGS option, the default usage is as follows:

Registers	**Default Usage**
0–10	Nonpreserved
11–15	Preserved

As an example, if the PUSHJ linkage-type is given without any linkage-option, then the resulting register usage is the following:

Registers	**Usage**
0	Stack pointer
1	Nonpreserved
2	Frame pointer
3	Value return register, nonpreserved
4–10	Nonpreserved
11–15	Preserved

Defaults for the JSYS Linkage-Type

For JSYS, the registers are used as follows, by default:

Registers	**Default Usage**
0	Preserved
5–15	Preserved
1–4	Nonpreserved

Defaults for the F10 Linkage-Type

For F10, the registers are used as follows, by default:

Registers	**Default Usage**
0	Value return register, nonpreserved
1–13	Nonpreserved

14	Argument pointer
15	Stack pointer

Observe that a frame pointer is not used.

Defaults for the PS_INTERRUPT Linkage-Type

For PS_INTERRUPT, the default register usage is determined in two steps: First, the defaults for the LINKAGE_REGS option are applied if the LINKAGE_ REGS option is not given; second, the defaults for all remaining registers are determined. The registers are used as follows, by default:

Registers	**Default Usage**
0	Preserved
1	Value return register, nonpreserved
2–12	Preserved
13	Frame pointer
14	Preserved
15	Stack pointer

Note that the value return register is specified but unused.

13.4.4. Semantics

The GLOBAL linkage-option can be used with both PUSHJ and F10 linkage-types. It is introduced in *Section 13.1, "Introduction to Linkage-Declarations"* and is discussed in detail in *Section 13.7, "Global Register Data Segments and Linkages"*.

The PORTAL linkage-option is used with the PUSHJ, F10, and PS_INTERRUPT linkage-types. When used in the definition of the linkage-attribute of a ROUTINE or GLOBAL ROUTINE declaration, it causes the first instruction of the code compiled for the routine to be a PORTAL instruction (JRST 1,. +1). The PORTAL instruction is used in the construction of certain kinds of execute-only programs. See the system hardware manuals for details.

The PRESERVE and NOPRESERVE linkage-options are described in *Section 13.1, "Introduction to Linkage-Declarations"*.

The LINKAGE_REGS option, used only with the PUSHJ and PS_INTERRUPT linkage-types, specifies the registers to be used for the stack pointer, frame pointer, and the value return register.

13.4.4.1. PUSHJ Linkage-Type

The PUSHJ linkage-type specifies a calling sequence in which the actual-parameters are passed on the stack without the use of an argument pointer. Unlike the F10 linkage-type, actual-parameters can also be passed in registers (as described in 13.1.1.3) and the LINKAGE_REGS option can be used to specify which registers are used for the stack pointer, frame pointer, and value return registers. For example:

```
LINKAGE
   DBL_PREC = PUSHJ( ; REGISTER=1, REGISTER=2):
                     LINKAGE_REGS(15,13,1)
                     NOPRESERVE(2,3,4,5)
                     PRESERVE(0,6,7,8,9,10,11,12,14);
```

The example defines linkage for a double-precision result in AC1 and AC2, with STANDARD locations (that is, the stack) reserved for an arbitrary number of inputs. Because AC1 is treated as an output-parameter-location, the routine should be NOVALUE.

The SKIP linkage modifier determines how PUSHJ returns to the calling-location. The following describes the skip-values used:

0	The routine returns to the calling-location plus 1 (this is the default skip-value).
1	The routine may return to the calling-location plus 1 or plus 2. The call value is zero (no skip) or 1 (skip).
2	The routine may return to the calling-location plus 1, 2, or 3. The call value is then zero, 1, or 2 respectively.

A nonzero skip-value must appear only in a valued routine, and a value-return register must be NOPRESERVE.

For ROUTINE declarations, the return-value is added to the saved PC value; therefore, the routine must not be NOVALUE.

The CLEARSTACK linkage modifier can be used only with PUSHJ. This option specifies that the actual-parameters (placed on the stack by a routine-call) will be removed from the stack by the called routine, instead of the calling routine. If the modifier is not specified, the parameters will not be removed from the stack by the called routine and become the responsibility of the caller. Be aware, however, that the number of actual-parameters used in the call must be exactly equal to the number of formal-parameters declared.

13.4.4.2. JSYS Linkage-Type

The JSYS linkage-type specifies a calling sequence in which actual-parameters are passed by register to TOPS–20 JSYS functions. For example:

```
LINKAGE
    SIN_LNKG = JSYS(REGISTER=1, REGISTER=2, REGISTER=3, REGISTER=4;
                    REGISTER=1, REGISTER=2, REGISTER=3 ) :SKIP(-1);
BIND ROUTINE
    SIN = %O'52' :SIN_LNKG;
```

The SIN routine reads a string from a specified source to the caller's address space using an inline JSYS instruction; parameters are passed by means of AC1-AC4.

The SKIP linkage modifier determines how the JSYS will return to the calling-location. The following describes the skip-values used:

-1	The instruction after the JSYS will be an ERJMP. The value of the call is zero if an error occurs; otherwise the value is a 1.
0	Control is returned to the next instruction; the value of the call is zero.
1	Control returns to the calling-location plus 1 or plus 2. The value of the call is zero (no skip) or 1 (skip).
2	Same as 1, except control also can return to the calling-location plus 3 (in which case, the value of the function is 2).

13.4.4.3. F10 Linkage-Type

The F10 linkage-type specifies a calling sequence in which input-actual-parameters are passed using an argument block (see *Section 13.1.1.1, "Special Purposes"*) whose address is contained in register 14. Register 15 is the stack pointer and register 0 is the value return register.

13.4.4.4. PS_INTERRUPT Linkage-Type

The PS_INTERRUPT linkage-type is similar to PUSHJ and compatible with TOPS–10 and TOPS–20 software interrupt (PSI) mechanisms; as such, a PS_INTERRUPT makes use of the DEBRK% JSYS and DEBRK. UUO exit mechanisms for TOPS–20 and TOPS–10. For example:

```
LINKAGE
    INTERRUPT = PS_INTERRUPT;
    ROUTINE PSI: INTERRUPT =
    BEGIN
    ...
    END;
```

Assuming a TOPS–20 compilation, the code expansion would be as follows:

```
PSI:     PUSH     SP, [ PSI36% ]   ;fake return PC to keep
                                   ;stack adjusted
         PUSH     SP, FP           ;[OPT] set up frame
         MOVE     FP, SP           ;[OPT]
         PUSH     SP, ...          ;[OPT] save necessary ACs
          ...
         POP      SP, ...          ;[OPT] restore saved ACs
         POP      SP, FP           ;[OPT] recover old FP
         ADJSP    SP, 1            ;Remove fake return PC
         DEBRK%                    ;Return to monitor
```

Notice that the expansion is exactly like that of a PUSHJ routine, the exception being that at routine entry the called routine places a dummy PC on the stack, and at routine exit the dummy PC is removed before the DEBRK% JSYS is executed. The environment is the same for TOPS–10, the only exception being that DEBRK.UUO is used to exit the routine.

A routine declared as a PS_INTERRUPT type must adhere to the following rules:

1. The routine must only be called by the PSI system.

2. The routine must only fetch from or assign to data segments which satisfy one of the following requirements:

   - A data-segment whose scope is limited to the body of the routine

   - A data-segment declared with a VOLATILE attribute

3. If an UNWIND can occur within the scope of the routine, a condition handler must be established by means of an ENABLE declaration within the routine.

When an UNWIND occurs, it is necessary that a DEBRK% JSYS, or DEBRK.UUO be executed to allow subsequent software interrupts to occur. To guarantee future interrupts the user must establish a condition handler in the PS_INTERRUPT linked routine. The BLISS–36 OTS uses this handler to ensure that the software interrupt system is re-enabled.

13.4.5. BLISS–36 Predeclared Linkage-Names

Four linkage-names are predeclared in every BLISS–36 module. These linkages are provided for compatible and transportable usage among the BLISS dialects. See *Section 13.5, "Common Predeclared Linkage-Names"* concerning such usage. The default linkage-name is BLISS36C. The predeclared linkage-names are defined as shown in the following declaration:

```
LINKAGE
    BLISS10 = PUSHJ,
    BLISS36C =
        PUSHJ:
            LINKAGE_REGS(15,13,1)
            NOPRESERVE(2,3,4,5)
            PRESERVE(0,6,7,8,9,10,11,12,14),
    FORTRAN_SUB = F10,
    FORTRAN_FUNC =
        F10: PRESERVE(2,3,4,5,6,7,8,9,10,11,12,13);
```

The BLISS10 linkage is provided for convenient interfacing with routines compiled by the BLISS–10 compiler. BLISS–10 is an older dialect of BLISS that is becoming obsolete. The definition of the BLISS10 linkage given here assumes that default register options are used by the BLISS–10 module.

The BLISS36C linkage is the default linkage for BLISS–36. The BLISS36C linkage can also be used for BLISS–10 routines that are compiled with the /Z qualifier of the BLISS–10 compiler.

13.5. Common Predeclared Linkage-Names

Two linkage-names are predeclared in all BLISS dialects, FORTRAN_SUB and FORTRAN_FUNC. In addition, the linkage-names BLISS and FORTRAN are predeclared in BLISS–16 and BLISS–32.

The complete semantics for these linkage-names is given in the earlier sections on the linkage-declaration for each dialect (see *Section 13.2.5, "BLISS–16 Predeclared Linkage-Names"* for BLISS–16, *Section 13.3.5, "BLISS–32 Predeclared Linkage-Names"* for BLISS–32, and *Section 13.4.5, "BLISS–36 Predeclared Linkage-Names"* for BLISS–36). This section summarizes the common characteristics that apply across dialects.

13.5.1. The BLISS Linkages

In BLISS–16 and BLISS–32, the BLISS linkage is the default linkage in the absence of any other specification. In BLISS–36, the default linkage is BLISS36C. The semantics associated with these linkages is given in *Section 12.4, "Ordinary-Routine-Declarations"* through *Section 12.7, "External-Routine-Declarations"*.

In light of the defaults, the way to obtain a compatible and transportable BLISS linkage in all dialects is to use no explicit linkage specification at all.

13.5.2. The FORTRAN Linkages

The FORTRAN-related linkages provide a compatible and transportable means to interface with FORTRAN compiled routines on each of the target systems. Use of the FORTRAN linkages is quite similar to use of the BLISS linkages with these exceptions:

- Each formal parameter must be assumed to contain a value that is an address. The body of the routine must be written appropriately. In BLISS– 32, this restriction can be relaxed through use of the %VAL built-in function of VAX FORTRAN.

- Each actual-parameter must be a value that is an address.

There are several FORTRAN linkages because, in the case of FORTRAN-10 on the DECsystem–10/20, FORTRAN-10 compiled SUBROUTINE subprograms use the machine registers in a different way than FORTRAN-10 compiled FUNCTION subprograms. This difference is reflected in the declarations for the FORTRAN_SUB and FORTRAN_FUNC linkage-names given for BLISS–36 in *Section 13.4.5, "BLISS–36 Predeclared Linkage-Names"*. There is no such difference for PDP–11 and VAX FORTRAN systems.

To obtain compatible and transportable interfacing to FORTRAN with all three BLISS dialects, do the following:

- Use the FORTRAN_SUB linkage-name in the declaration of any routine which is to be used as a FORTRAN SUBROUTINE subprogram.

  This applies to all EXTERNAL ROUTINE declarations, for example, regardless of whether the routine is actually written in BLISS or FORTRAN. This also applies, obviously, to the ROUTINE or GLOBAL ROUTINE declaration if the routine is written in BLISS. In both cases, it is also highly desirable to use the NOVALUE attribute as well.

- Use the FORTRAN_FUNC linkage-name in the declaration of any routine which is to be used as a FORTRAN FUNCTION subprogram.

  As with the FORTRAN_SUB linkage, this applies to EXTERNAL ROUTINE declarations as well as to ROUTINE and GLOBAL ROUTINE declarations.

If compatible and transportable interfacing to only PDP–11 and VAX FORTRAN systems is desired, then the FORTRAN linkage-name can be used for both SUBROUTINE and FUNCTION subprograms in BLISS–16 and BLISS–32.

13.6. Linkage-Functions

Linkage-functions are executable-functions (see *Section 5.2, "Executable-Functions"*) that provide specialized information about the actual-parameters used to call a routine. For example, linkage-functions can be used to code a routine that can be called with different numbers of actual-parameters in different routine-calls.

13.6.1. Common Linkage-Functions

There are three common BLISS linkage-functions: ACTUALCOUNT, ACTUALPARAMETER and ARGPTR. These functions can be used with all of the FORTRAN-related predeclared linkages in all BLISS dialects. They can also be used with some of the BLISS–related predeclared linkages.

13.6.1.1. Definition

The common linkage-functions are defined as follows:

- ACTUALCOUNT()

  Restriction: Must be declared BUILTIN within the body of a routine whose linkage-attribute is defined with certain linkage-types. The linkage-types, and the predeclared linkages that are consequently permitted, are as follows:

Dialect	Linkage-Type	Predeclared Linkages
BLISS–16	CALL	FORTRAN

		FORTRAN_SUB FORTRAN_FUNC
BLISS–32	CALL	BLISS FORTRAN FORTRAN_SUB FORTRAN_FUNC
BLISS–36	F10	FORTRAN_SUB FORTRAN_FUNC

Value: Return the number of actual-parameters passed to the routine using STANDARD parameter-locations; parameters passed using REGISTER parameter-locations are not included in the returned value.

For the predeclared linkages in all dialects, all parameters are passed using STANDARD parameter-locations and, consequently, ACTUALCOUNT returns the number of actual-parameters.

- ACTUALPARAMETER(i)

  Restriction: The first restriction for ACTUALPARAMETER is the same as for ACTUALCOUNT above.

  The value of i must be in the range 1 to ACTUALCOUNT().

  Value: Return the value of the ith actual-parameter that was passed using STANDARD parameter-locations; parameters passed using REGISTER parameter-locations are not obtainable with this function.

  For the predeclared linkages in all dialects, all actual-parameters are passed using STANDARD parameter-locations, and, consequently, ACTUALPARAMETER(i) returns the value of the ith actual-parameter.

- ARGPTR()

  Restriction: The restriction for ARGPTR is the same as for ACTUALCOUNT above.

  Value: Return the address of the argument block.

13.6.1.2. Examples

The use of the linkage-functions permits routines to be written in a more general way. Consider, for example, a generalization of the routine AVERAGE3 (*Section 12.4.5, "Pragmatics"*), which accepts three parameters, to the routine AVERAGE, which accepts any number of parameters:

```
ROUTINE AVERAGE =
    BEGIN
    BUILTIN
        ACTUALCOUNT,
        ACTUALPARAMETER;
    LOCAL
        L;
    L = 0;
    INCR I FROM 1 TO ACTUALCOUNT() DO
        L = .L + ACTUALPARAMETER(.I);
    .L/ACTUALCOUNT()
    END;
```

Some calls on the routine AVERAGE and the value of these calls are as follows:

Call	Value
AVERAGE(1,2,3)	2
AVERAGE(2,4,6,8,10)	6
AVERAGE(8)	8
AVERAGE()	??? (Invalid)

In some cases, a routine has a fixed and variable set of parameters. For example, consider the following routine, which calculates the difference between an expected value (the fixed part) and the average of a set of values (the variable part):

```
ROUTINE DELTA_AVERAGE(EXPECTED) =
    BEGIN
    BUILTIN
        ACTUALCOUNT,
        ACTUALPARAMETER;
    LOCAL
        L;
    L = 0;
    INCR I FROM 2 TO ACTUALCOUNT() DO
        L = .L + ACTUALPARAMETER(.I);
    .EXPECTED - .L/(ACTUALCOUNT()-1)
    END;
```

Some calls on the routine DELTA_AVERAGE follow:

Call	Value
DELTA_AVERAGE(3,1,2,3)	1
DELTA_AVERAGE(6,2,4,6,8,10)	0
DELTA_AVERAGE(7)	??? (Invalid)
DELTA_AVERAGE()	??? (Invalid)

Observe in this example that explicit formal-parameters are not distinct from the parameters accessed by the linkage-functions. Specifically, .EXPECTED is equivalent to ACTUALPARAMETER(1). Consequently, the loop initial value is 2, not 1, and the divisor in the next to last line is ACTUALCOUNT()-1, not ACTUALCOUNT().

The ARGPTR linkage-function returns the address of the argument block of a routine-call. In some cases the argument block address passed in the argument pointer register may not be left in that same register throughout the execution of the called routine. For example, in BLISS–36 this is usually done in the code compiled for a routine with the F10 linkage-type that calls another routine which also has the F10 linkage-type. The ARGPTR function provides a compatible means to obtain the address of the argument block in all dialects.

13.6.2. BLISS–16 and BLISS–32 Linkage-Functions

The NULLPARAMETER linkage-function (in BLISS–16 and BLISS–32 only) tests a parameter position of a call from a FORTRAN routine and returns true if the actual-parameter is a null or omitted parameter. See the PDP–11 and VAX FORTRAN manuals for a description of null and omitted parameters.

The NULLPARAMETER linkage-function is defined as follows:

NULLPARAMETER(i)

Restriction: If i is not a formal-name then it is interpreted as an expression and the value of i must then be greater than or equal to 1. The linkage-type and predeclared linkages that are permitted are as follows:

Dialect	Linkage-Type	Predeclared Linkages
BLISS–16	CALL	FORTRAN FORTRAN_SUB FORTRAN_FUNC
BLISS–32	CALL	BLISS FORTRAN FORTRAN_SUB FORTRAN_FUNC

Value: If i is a formal-name and the corresponding actual-parameter tested is null, or omitted, a value of 1 is returned; otherwise, a zero is returned. If i is an expression, a value of 1 is returned when i is greater than the number of actual-parameters. A value of 1 is also returned if i is not greater than the number but the ith actual-parameter has the value –1 in BLISS–16 or 0 in BLISS–32; otherwise, a zero is returned.

13.7. Global Register Data Segments and Linkages

A *global register data segment* is a data segment that is created and allocated in a given register in one routine and can be made available for use in other routines that it calls. Global register data segments are identified by name and both the calling and called routine must agree that a particular data segment is available.

A GLOBAL REGISTER declaration (*Section 10.8, "Global-Register-Declarations"*) causes a global register data segment to be allocated. A global register data segment is a local data segment just like an ordinary register data segment – it is created on entry to the block in which it is contained and released on exit from that block. However, unlike an ordinary register data segment, a global data segment is available in called routines under certain circumstances.

To pass a global register data segment to a called routine, the linkage-attribute for the called routine must contain the name of the data segment and its register assignment in its GLOBAL linkage-option. There may be more global register data segments available at a call than are given in the linkage for the call; however, every global register data segment given in the linkage must be available at the call. Only those global register data segments given in the linkage are available in the called routine.

An EXTERNAL REGISTER declaration (*Section 10.9, "External-Register-Declarations"*) specifies that a global register data segment created in a calling routine is available for use. The declared name must be given in the linkage; however, not all global register data segments given in the linkage need be declared in an EXTERNAL REGISTER declaration.

The linkage-attribute forms a bridge between calling and called routines. Consider the use of the global register data segment GRDS in the following example:

```
%IF %BLISS(BLISS16) OR %BLISS(BLISS32)
    %THEN
    LITERAL
```

```
        GRDS_REG = 1 ;
    LINKAGE
        BRIDGE =
            %BLISS16(JSR: GLOBAL(GRDS = GRDS_REG))
            %BLISS32(CALL: GLOBAL(GRDS = GRDS_REG));
    %ELSE     ! For BLISS--36
    LITERAL
        GRDS_REG = 6 ;
    LINKAGE
        BRIDGE = PUSHJ:
                    LINKAGE_REGS(15,13,1)
                    NOPRESERVE(2,3,4,5)
                    PRESERVE(0,7,8,9,10,11,12,14)
                    GLOBAL(GRDS = GRDS_REG);
%FI
FORWARD ROUTINE
    ROUT2: BRIDGE NOVALUE;
ROUTINE ROUT1 =
    BEGIN
    GLOBAL REGISTER
        GRDS = GRDS_REG;
    GRDS = 0;
    ROUT2();
    .GRDS
    END;
ROUTINE ROUT2: BRIDGE NOVALUE =
    BEGIN
    EXTERNAL REGISTER
        GRDS;
    GRDS = .GRDS + 1;
    END;
```

First, the literal-name GRDS_REG is bound to either the value 1 or the value 6, depending upon the compiler used for the compilation. This literal value is used to specify a register number in several subsequent declarations. The conditional-compilation constructs used in this example are described in *Chapter 15, "Lexical Functions"* and *Chapter 16, "Macros"*.

Next, the name BRIDGE is defined as a linkage-name with the global register data segment GRDS. This declaration also depends upon the compiler used for the compilation. Note that the definition of BRIDGE for BLISS–36 matches the default BLISS36C linkage except for the GLOBAL option, and thus is compatible with the default linkage. Then, the forward-routine-declaration for ROUT2 uses the linkage-attribute BRIDGE. The calling routine ROUT1 allocates the global register data segment GRDS and sets it to 0. Observe that ROUT1 does not need any special linkage-attribute in order to create the global register data segment. ROUT1 then calls the routine ROUT2. ROUT2 increments the value of the global register data segment, and returns. The value of routine ROUT1 is the value of the global register data segment, 1.

Because the information about the global register data segment is supplied by the linkage-attribute BRIDGE, the compiler can perform several consistency checks to verify that the global register data segment is being used correctly. In the above example, the compiler knows that ROUT2 uses a global register data segment and can, therefore, check that a call on that routine occurs within the scope of the global register declaration. Further, the compiler can check that the external register declaration for GRDS is within a routine with a linkage-attribute for the global register data segment GRDS.

A global register data segment is a register that is, by convention, reserved for a particular use by a set of routines that function together as a package. For example, consider a file maintenance package. Typically, such a package consists of interface routines and internal routines. The interface routines

establish the function to be performed by the file maintenance package (for example, open, insert, and so on) and set up the appropriate environment. The internal routines perform the basic processing within the environment established by the interface. Part of that environment is often the establishment of one or more global register data segments.

To describe some of the elements of a file maintenance package, the following provides a simplified version of such a system.

The module consists of two global routines, VECMAXMIN and VECMAXMINAVG, each of which uses two additional internal routines. Both VECMAXMIN and VECMAXMINAVG are written to be callable from FORTRAN. Each actual-parameter to these routines must be the address of the desired FORTRAN variable or array.

The first routine, VECMAXMIN, is called with the first parameter giving the base of an integer vector, and the second parameter giving the number of elements in the vector. The maximum value encountered in the vector is returned via the third parameter, while the minimum value is returned via the fourth parameter. The value of the routine is the difference between the maximum and minimum.

The second routine, VECMAXMINAVG, is called with two parameters which are the same as the first two parameters of VECMAXMIN. Its value is the average of the maximum and minimum elements of the array.

The internal routine VECMAX1 searches a vector and returns the maximum value; and similarly, the internal routine VECMIN1 returns the minimum value. Routines VECMIN1 and VECMAX1 each receive their two parameters as global register data segments, in registers that are appropriate for the respective, dialect-specific linkage definitions. See the guidelines given further on concerning the preferred choice of registers for each target system.

```
MODULE VECOPS(IDENT='03') =
BEGIN

LITERAL
    VECREG = %BLISS16(1)
             %BLISS32(11)
             %BLISS36(12),
    LENREG = %BLISS16(2)
             %BLISS32(10)
             %BLISS36(11);
LINKAGE
    BLISSTWOREG =
        %BLISS16(JSR:)
        %BLISS32(CALL:)
        %BLISS36(PUSHJ: LINKAGE_REGS(15,13,1)
                        NOPRESERVE(2,3,4,5)
                        PRESERVE(0,7,8,9,10,14))
            GLOBAL(VEC = VECREG, LEN = LENREG);

FORWARD ROUTINE
    VECMAXMIN: FORTRAN_FUNC,
    VECMAXMINAVG: FORTRAN_FUNC,
    VECMAX1: BLISSTWOREG,
    VECMIN1: BLISSTWOREG;

GLOBAL ROUTINE
    VECMAXMIN(VECADR,LENADR,MAXADR,MINADR): FORTRAN_FUNC =
    BEGIN
```

```
    GLOBAL REGISTER
        VEC = VECREG : REF VECTOR,
        LEN = LENREG;

    ! Initialize global registers
    !
    VEC = .VECADR;
    LEN = ..LENADR;
    ! Main code
    !
    .MAXADR = VECMAX1();
    .MINADR = VECMIN1();
    ..MAXADR-..MINADR
    END;

GLOBAL ROUTINE VECMAXMINAVG(VECADR,LENADR): FORTRAN_FUNC =
    BEGIN

    GLOBAL REGISTER
        VEC = VECREG : REF VECTOR,
        LEN = LENREG;

    VEC = .VECADR;
    LEN = ..LENADR;

    (VECMAX1() - VECMIN1())/2
    END;
ROUTINE VECMAX1: BLISSTWOREG =
    BEGIN

    EXTERNAL REGISTER
        VEC: REF VECTOR,
        LEN;

    LOCAL
        MAXX;

    MAXX = .VEC[0];
    DECR J FROM .LEN-1 TO 1 DO
        MAXX = MAX(.MAXX,.VEC[.J]);

    .MAXX
    END;
ROUTINE VECMIN1: BLISSTWOREG =
    BEGIN

    EXTERNAL REGISTER
        VEC: REF VECTOR,
        LEN;

    LOCAL
        MINN;

    MINN = .VEC[0];
    DECR J FROM .LEN-1 TO 1 DO
        MINN = MIN(.MINN,.VEC[.J]);
    .MINN
    END;
```

```
END
ELUDOM
```

13.7.1. Discussion

GLOBAL REGISTER and EXTERNAL REGISTER declarations in combination with linkage-definitions that include a GLOBAL linkage-option provide a controlled means to extend the scope of a register data segment from one routine into another routine. The restrictions help assure that this unusual dynamic extension of register scope is clearly documented and unlikely to be a source of error because of hidden effects.

The use of global register data segments provides two optimization benefits. First, both the called and the calling routines benefit from code efficiency that results from the use of a register instead of a temporary (stack) location to hold the parameter value during the call. Second, the calling routine benefits from the fact that the global register value is still available in the same register after return from the called routine. The saving and restoring of the register contents is not required around the call.

The same conventions can (and must) be used to share register data segments between nested routine definitions. In this case, the convention allows the inner routine to access a local data segment of the outer routine in an efficient manner. This capability is sometimes called "up-level addressing" in other languages and often requires complex and inefficient code. Observe, however, that there is no particular advantage to writing the called routine as a nested routine. Indeed, the convention works equally well between routines in separately compiled modules.

Using global registers is a useful optimization technique. However, when using this technique, you must ensure that independently developed parts of the program do not inadvertently use register assignments that would be in conflict when the parts are brought together. Global registers are not subject to the normal optimization strategies of the compiler and, consequently, may lead to worse, rather than better, code quality if too many are used.

13.7.2. Guidelines for BLISS–16

The many restrictions concerning the use of LINKAGE declarations and global register data segments are necessary to ensure proper management of the machine registers at all times.

Two guidelines are particularly recommended:

- The value return register should always be specified as nonpreserved (which is the default). This will avoid the special restrictions related to this register.
- When planning the allocation of global register data segments, use contiguous registers beginning with register 1; for example, if two registers are needed, use registers 1 and 2.

Note that, because the PDP–11 has very few locally usable registers (relative to other target systems), the allocation of even one register as global over a large span of code will very likely decrease overall code quality.

13.7.3. Guidelines for BLISS–32

The many restrictions concerning the use of LINKAGE declarations and global register data segments are necessary to ensure proper management of the machine registers at all times, especially during condition handling (see *Chapter 17, "Condition Handling"*).

One restriction in particular deserves special consideration when JSB routines and global register data segments are used together: If a call to a routine with JSB linkage-type occurs in the scope of a global register data segment, then the given register-number of the data segment must be given in either a GLOBAL linkage-option or a NOTUSED linkage-option of the linkage of the called routine.

That is, if a global register data segment is active at the point of a call to a JSB routine, the only permitted use of the register in the JSB routine is as a global register data segment; if not used that way, it must not be used at all.

Some service routines in the VMS Run-Time Library (RTL) use JSB linkage. By convention, these routines use a contiguous group of registers, none of which are preserved, starting at register number 0. In light of this convention, and the above restrictions, the following two guidelines are suggested:

- When specifying the linkage of a routine with JSB linkage, give the locally usable registers as contiguous lower numbered registers starting at zero. Keep the set of locally usable registers as small as possible consistent with acceptable code quality.
- When planning the allocation of global register data segments, use contiguous higher-numbered registers, that is, 11, 10, 9, and so on.

A reasonable strategy is to divide the registers into groups so that JSB routines never locally use more than, for example, registers 0 through 7 and global register data segments are always specified in registers 8 through 11. This guarantees that no conflicts will arise in using JSB routines and global register data segments together.

One additional guideline is strongly recommended: Registers 0 and 1 should always be specified as nonpreserved (which is the default). This will avoid the error-prone special restrictions related to condition handling (see *Section 13.3.2, "Restrictions"*).

13.7.4. Guidelines for BLISS–36

The many restrictions concerning the use of LINKAGE declarations and global register data segments are necessary to ensure proper management of the machine registers at all times.

Two guidelines are particularly recommended:

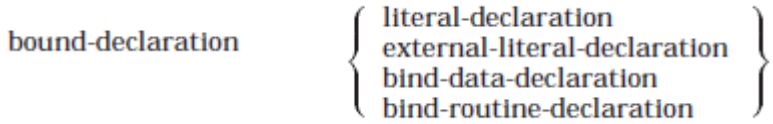
- The value return register should always be specified as nonpreserved (which is the default). This will avoid the special restrictions related to this register.
- When planning the allocation of global register data segments, use the highest-numbered contiguous set of registers available. For example, 12, 11, 10, 9, and so on when using the BLISS36C type register conventions.

Chapter 14. Binding

Bound-declarations are different from most of the declarations discussed thus far because a bound-declaration defines a name in terms of other names and values. Bound-declarations do not involve the allocation of storage. Instead, they provide a name for a constant value, or an additional name and sometimes a different interpretation for existing storage.

A bound-declaration defines a name. The definition of a name consists of its scope, its value, and its attributes. The scope and attributes are determined in the usual way. However, the value of the name defined in the bound-declaration is determined from the value of an expression.

A name can be defined by a bound-declaration to be a literal-name, a data-name, or a routine-name. The syntax diagram for bound-declarations is as follows:



The syntax and semantics for each kind of bound-declaration are given in the following sections.

14.1. Literal-Declarations

A literal-declaration is used to define a name whose value is determined by a constant expression. After a name is defined in this way, it can be used to designate the constant expression.

A literal-declaration can contribute to the readability of a program. An example of this usage follows:

```
LITERAL
    CAPACITY = 25;
```

This declaration allows the following assignment to be written:

```
STUDENTS = .ROOMS * CAPACITY
```

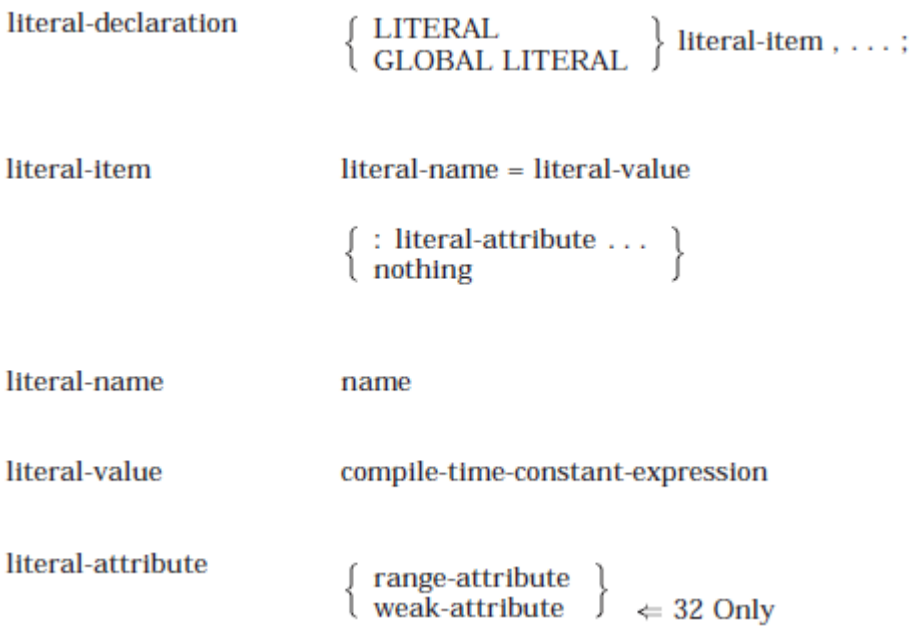
In this expression, STUDENTS and ROOMS are data segment names and CAPACITY is the literal name declared above. The use of the literal-declaration makes clear the significance of the value 25.

A literal-declaration is especially useful for defining a constant value that is used at several different places in a program. In the event that a different version of the program requires a different value for the constant value, the change can be made in just one place; namely, in the literal-declaration. An example of this usage follows:

```
LITERAL
    BUFFERSIZE = 266;
```

It is assumed that the size of the buffer changes from time to time and that this value is involved in computations throughout the program. A change in the value of BUFFERSIZE in this declaration automatically changes the value of all the occurrences of BUFFERSIZE within the program.

14.1.1. Syntax



14.1.2. Restrictions

The value, *n*, of the bit-count expression in the range-attribute must lie in the range $1 < n <$ %BPVAL.

The literal-value must be representable in the given number of bits.

BLISS–32 ONLY

The WEAK attribute can be specified only in a GLOBAL LITERAL declaration.

14.1.3. Defaults

If a range-attribute is not specified, then SIGNED(%BPVAL) is assumed.

14.1.4. Semantics

A literal-declaration is processed by the compiler as follows:

1. The literal-value expression is evaluated.
2. The range-attribute is used to validate the representation of the literal-value. The bit-count expression is evaluated and the value obtained in step 1 is checked to verify that it can be represented as a SIGNED or UNSIGNED value in the number of bits specified.
3. If the literal-declaration is GLOBAL or GLOBAL with the weak-attribute (BLISS–32 only), then the appropriate indicators are set for the linker.
4. The literal-name is associated with the value represented in step 2. Wherever the literal-name appears in the module, it is replaced by its associated value.

14.1.5. Predeclared Literals

The following literal-names are predeclared in BLISS:

Name	BLISS–16 Value	BLISS–32 Value	BLISS–36 Value	Significance
%BPVAL	16	32	36	Bits per BLISS value (fullword)
%BPUNIT	8	8	36	Bits per smallest addressable unit
%BPADDR	16	32	18 or 30	Bits per address value
%UPVAL	2	4	1	Addressable units per BLISS value (%BPVAL divided by %BPUNIT)

The value of %BPADDR in BLISS–36 is determined by the cpu-option setting of the ENVIRONMENT module-switch (*Section 19.2, "Module-Switches"*); see the *BLISS–36 User's Guide* for target-system environment information.

These predeclared names can be used to enhance the transportability of a program from one target system to another. See the appropriate BLISS user manual for further information.

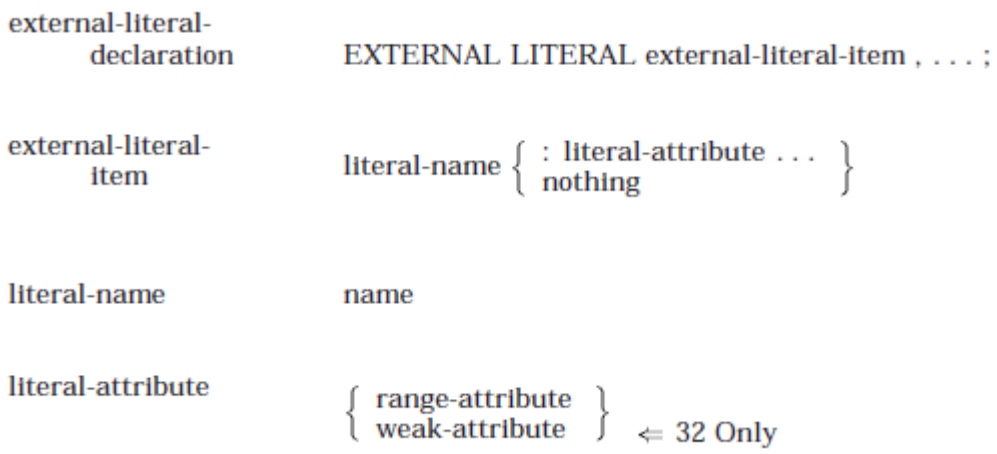
14.2. External-Literal-Declarations

An external-literal-declaration gives a list of literal-names that are declared in other, separately compiled, modules. When the program that contains these modules is linked, the value of the external-literal-names is determined.

External-literal-declarations are useful for providing mnemonic names for constant expressions that are common to the modules of a program. An example of an external-literal declaration follows:

```
EXTERNAL LITERAL
    BLKSIZ: SIGNED(8);
```

14.2.1. Syntax



14.2.2. Restrictions

A name must not be declared EXTERNAL LITERAL unless it is declared GLOBAL LITERAL in some other block or module of the same program. This restriction does not apply, however, to a name that is declared with the weak-attribute in BLISS–32 (see *Section 9.14, "The Weak-Attribute – BLISS–32 Only"*).

The range-attribute for an EXTERNAL literal-name must accommodate the value given for the literal in its GLOBAL literal-declaration. For further discussion, see *Section 9.10, "The Novalue-Attribute"*.

14.2.3. Defaults

If a range-attribute is not given, then SIGNED(%BPVAL) is assumed.

14.2.4. Semantics

An external-literal-declaration is processed by the compiler as follows:

1. Each name in the list is identified as an EXTERNAL literal-name.

2. If the WEAK attribute is specified, an indicator is provided for the linker (BLISS–32 only).

14.3. Bind-Data-Declarations

A bind-data-declaration is used to define another name for a data segment, or part of a data segment, that already exists. The bound name can have different attributes and can therefore depart from the original interpretation of the data segment. An example of a bind-data-declaration appears in the following program fragment:

```
OWN
    ALPHA: VECTOR[20];
BIND
    A = ALPHA[8];
...
INCR I FROM 0 TO 20 DO
    ALPHA[.I] = .ALPHA[.I] * .A;
```

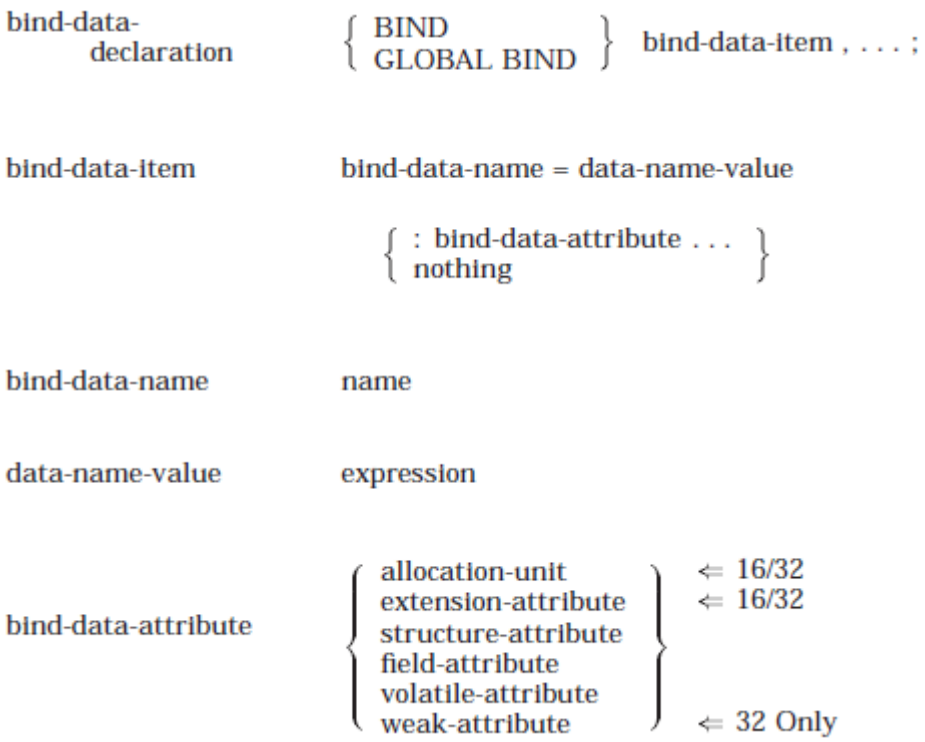
The name A is defined by the bind-data-declaration to be a fullword scalar with the same address as the ninth element of the vector ALPHA. A reference to A, therefore, is equivalent to, but more concise than, a reference to ALPHA[8].

In the example just given, the value of A can be determined at the time the program is linked since the address of the ninth element of the vector ALPHA is known at link time. An example of a binding that cannot be determined at link time is as follows:

```
BIND B = ALPHA[2*.J-1]
```

The contents of J is not known at link time and so the binding of B is deferred to execution time. Specifically, the binding occurs just before the evaluation of the block in which the declaration appears. The introduction of the name B can be efficient because no matter how often B is used during the evaluation of the block, the expression 2*.J-1 is evaluated only once.

14.3.1. Syntax



14.3.2. Restrictions

The data-name-value expression must be the address of a data segment that can be accessed within the scope of the declaration.

The data-name-value expression must be a link-time constant expression if the declaration begins with GLOBAL or the declaration is at the outermost level of a module (and is not, therefore, contained in a routine-declaration).

The data-name-value expression in a GLOBAL bind-data-declaration is limited to a restricted subset of link-time constant expressions, in that it must not contain a name declared EXTERNAL, EXTERNAL ROUTINE, or EXTERNAL LITERAL unless that name is an operand of a compile-time constant expression (see *Section 7.1.2, "Restrictions"*). Furthermore, the data-name-value expression must not contain a name declared BIND, GLOBAL BIND, BIND ROUTINE, or GLOBAL BIND ROUTINE unless the definition of that name satisfies this same restriction.

A structure-attribute must not appear in a declaration that has an allocation-unit or an extension-attribute (BLISS–16/32 only).

A field-attribute can appear only in a declaration that has a structure-attribute.

A weak-attribute can appear only in a GLOBAL bind-data-declaration (BLISS–32 only).

14.3.3. Defaults

If an allocation-unit is not given, fullword allocation is assumed (BLISS–16/32 only).

If a structure-attribute is not given, the name is assumed to be a scalar.

14.3.4. Semantics

A bind-data-declaration is processed as follows:

1. The bind-data-name is associated with the attributes given either explicitly or by default in the declaration.

2. The value of the bind-data-name is determined. The time of evaluation depends on the kind of data-name-value expression given. If the expression is not a link-time constant expression, it is evaluated just prior to the evaluation of the immediately containing block.

14.4. Bind-Routine-Declarations

A bind-routine-declaration is used to define another name for an existing routine. After a routine-name is defined in this way, it can be used in the scope of the bind-routine-declaration either by itself to designate the value of the routine-name or with a parenthesized list of parameters to indicate a call on the routine.

An example of a bind-routine-declaration follows:

```
BIND ROUTINE CALC = CALCULATION4;
```

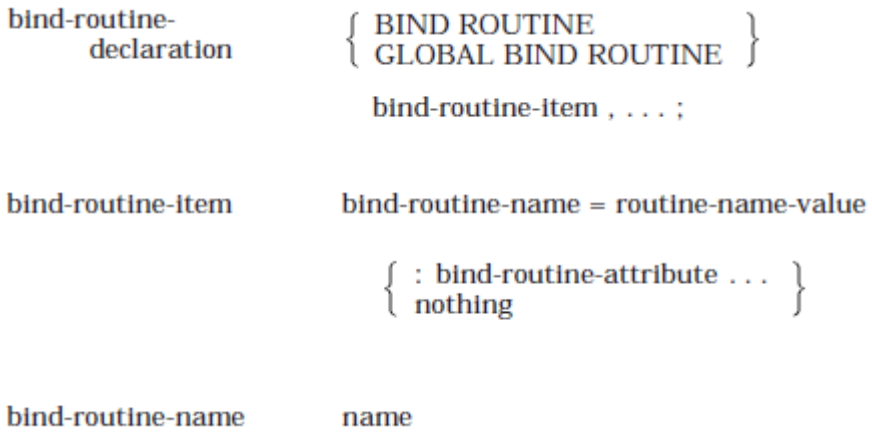
It is assumed that `CALCULATION4` is the name of a routine that is declared elsewhere, and under this assumption, the value of CALC can be determined at link time.

Another example of a bind-routine-declaration follows:

```
BIND ROUTINE SR = (IF .A LSS 0 THEN SNEG ELSE SPOS);
```

It is assumed that `SNEG` and `SPOS` are names of routines that are declared elsewhere. Because the expression to the right of the equal sign (=) operator is not a link-time constant expression, the value of SR is determined just before each evaluation of the block that contains the declaration.

14.4.1. Syntax



```
routine-name-value      expression

bind-routine-
      attribute         { novalue-attribute
                        { linkage-attribute
                        { weak-attribute      } ⇐ 32 Only
```

14.4.2. Restrictions

The value of the routine-name-value expression must be the address of a routine that can be called within the scope of the bind-routine-declaration.

The routine-name-value expression must be a link-time constant expression if the declaration begins with GLOBAL or the declaration is at the outermost level of a module (and is not, therefore, contained in a routine-declaration).

The routine-name-value expression in a GLOBAL bind-routine-declaration is limited to a restricted subset of link-time constant expressions, in that it must not contain a name declared EXTERNAL, EXTERNAL ROUTINE, or EXTERNAL LITERAL unless that name is an operand of a compile-time constant expression (see *Section 7.1.2, "Restrictions"*). Furthermore, the routine-name-value expression must not contain a name declared BIND, GLOBAL BIND, BIND ROUTINE, or GLOBAL BIND ROUTINE unless the definition of that name satisfies this same restriction.

The WEAK attribute must be given only with a GLOBAL bind-routine-declaration (BLISS–32 only).

14.4.3. Default

If a linkage-attribute is not given and the bind-routine-declaration is in the scope of a LINKAGE switch, then the default linkage-attribute is the linkage-name given in the linkage-switch (see *Section 18.2, "Switches-Declarations"* and *Section 19.2, "Module-Switches"*). Otherwise, the default linkage-attribute is the predeclared linkage-name BLISS in BLISS–16 and BLISS–32, or BLISS36C in BLISS–36.

14.4.4. Semantics

A bind-routine-declaration is processed as follows:

1. The bind-routine-name is associated with the attributes given either explicitly or by default in the declaration.

2. The value of the bind-routine-name is determined. The time of evaluation depends on the kind of routine-name-value expression given. If the expression is not a link-time constant expression, it is evaluated just before the evaluation of the immediately containing block.

Chapter 15. Lexical Functions

BLISS provides two groups of features that are concerned with the compile-time processing of a module: *lexical-functions*, described in this chapter, and *macros*, described in *Chapter 16, "Macros"*. Lexical functions and macros are closely related and share many common concepts and mechanisms. Consequently, the introduction to this chapter considers both together in an integrated way and lays the foundation for the description of macros in the next chapter.

The lexical-functions perform basic operations on the text of the module; for example, the %STRING lexical-function gathers several lexemes into a single quoted-string lexeme, and the %CHARCOUNT lexical-function counts the characters in a given quoted-string. The example material in this chapter includes both lexical-functions and macros, because in practical use these two features are usually intertwined.

Closely related to the lexical-functions is the *lexical-conditional*, which permits you to indicate that a portion of a program is to be included or omitted depending on the outcome of a given compile-time test. Another related facility is the *compile-time-declaration*, which declares names whose values can be changed during compilation and which can control macro-expansion.

All these facilities depend on *lexical processing*, which is the first step in the compilation of a module. During lexical processing, lexemes are formed and interpreted, names are associated with their declarations, and the various kinds of lexical constructs are processed.

The first section introduces lexical processing and the second section gives the quotation conventions. The next four sections describe lexical-expressions, lexical-functions (in general and in particular), and lexical-conditionals. The final section describes the compile-time-declaration.

15.1. Introduction to Lexical Processing

The compilation of a module begins with *lexical processing*, which divides the module into lexemes, binds names to their associated declarations, and expands macro-calls.

15.1.1. From Characters to Lexemes

A module is supplied to the compiler as a sequence of characters and linemarks. As the module is processed, the characters and linemarks are collected to form lexemes. The various kinds of lexemes are described in *Chapter 2, "Lexical Definitions and Syntax Notation"*.

As an example of conversion to lexemes, consider the following module:

```
MODULE EX =
BEGIN
GLOBAL
    X: VECTOR[1024];
END
ELUDOM
```

This module is presented to the compiler as a source file composed of the following characters:

```
M, O, D, U, L, E, blank, E, X, blank, =, linemark,
B, E, G, I, N, linemark,
G, L, O, B, A, L, linemark,
```

```
blank, blank, blank, blank, X, :,
blank, V, E, C, T, O, R, [, 1, 0, 2, 4, ], ;, linemark,
E, N, D, linemark,
E, L, U, D, O, M, linemark
```

As the module is read by the compiler, it is converted into the following list of 14 lexemes:

```
MODULE, EX, =,
BEGIN,
GLOBAL,
X, :, VECTOR, [, 1024, ], ;,
END,
ELUDOM
```

It is the lexemes that are important in BLISS, not the individual characters, and in the remainder of this chapter, modules are discussed as sequences of lexemes. That is, the division of modules into lexemes is taken for granted.

15.1.2. Lexeme-by-Lexeme Processing

The compiler works on one lexeme at a time. That is, the compiler does not read a new lexeme until it has done everything it can with the portion of the module it has already seen. This lexeme-by-lexeme processing is a fundamental characteristic of BLISS. For example:

```
OWN
    ALPHA;
ALPHA = 2;
```

When the compiler encounters this fragment, it is already in the midst of a module. Assume that the compiler has already encountered, in an outer block, a declaration of ALPHA as a literal-name. The first lexeme in the fragment is OWN. When the compiler reads this lexeme, it recognizes that the next lexeme will be a new declaration of some name, and it prepares for that situation.

The second lexeme is ALPHA. Although ALPHA is already declared, the compiler treats this occurrence of ALPHA as a new, overriding declaration of ALPHA.

The third lexeme is a semicolon. When the compiler reads this lexeme, it knows that the declaration is complete. Therefore, the compiler fills in the various defaults for ALPHA, providing a complete declaration for the name.

The fourth lexeme is another occurrence of ALPHA. Because of the context, the compiler knows that this occurrence of ALPHA is a use of the name rather than another declaration. Because the compiler is working on one lexeme at a time, it has the full declaration of ALPHA ready to apply to this use of ALPHA. And that is the main point of this example.

The lexeme-by-lexeme processing of a BLISS program is quite natural and obvious for simple modules, such as the example just given. However, in more complicated cases, there may be more than one "obvious" way to interpret a module, and the lexeme-by-lexeme rule must be invoked to determine what actually happens.

15.1.3. Binding

Every identifier that is not a reserved keyword can be used as a name. When an identifier is used as a name, it must be declared; that is, it must be associated with a declaration. Declarations can be implicit

(supplied by the compiler) or explicit (written by the programmer). The process of associating a given use of a name with a declaration is called *lexical binding*. The process of associating a declaration of a name with a storage address is also called binding, as discussed in *Section 1.4, "The Main Features of BLISS"*. Binding in this sense, however, is not a concern of this chapter.

In some cases, there is more than one way to lexically bind a name. Consider the following example:

```
LITERAL
    ABS = 0;
...
ROUTINE ALPHA(X): NOVALUE =
    BEGIN
    LOCAL
        ABS;
    ABS = ..X+1;
    .x = .ABS*..X;
    END;
```

In this example, there are three declarations of ABS. First, ABS is implicitly declared as the name of the absolute value executable function, as described in *Chapter 5, "Computational Expressions"*. Second, ABS is explicitly declared as LITERAL on the second line. Third, ABS is explicitly declared as LOCAL within the routine-declaration. According to the rules for scoping given in *Section 8.2, "Declarations"*, the use of ABS in the assignment to .X is bound to the third (and most recent) declaration of ABS.

15.1.4. Expansion

BLISS includes a facility for defining and using macros. Macros have names and the names are defined and given by declarations, just like other BLISS names. Thus, the macro facility is an integral part of the BLISS language.

A *macro-declaration* associates a sequence of lexemes, a *macro-body*, with a macro-name. Within the scope of the macro-declaration the macro-name can be used in a *macro-call*. During compilation, each macro-call is replaced by a copy of the macro-body.

A macro can be *parameterized*; that is, each macro-call can supply *actual-parameters* that are substituted for *formal-names* in the macro-body.

When the compiler encounters a macro-call, it first reads through the call itself, collecting and processing the actual-parameters. Then the compiler replaces the macro-call by its *expansion*. The expansion is a modified copy of the macro-body that is given in the declaration of the macro.

A simple example follows:

```
MACRO
    PROD(X) = (((X)+1)*((X)-1)) %;
...
B = PROD(2*A);
```

Here, the macro-call is `PROD(2*A)` and the macro-body is `((X)+1)*((X)-1)`. After the macro-call is expanded, the assignment to `B` becomes the following:

```
B = (((2*A)+1)*((2*A)-1));
```

The term "expansion" reflects the fact that macros are often used as a short way to express a long construct. Indeed, in the example above, the expansion is considerably longer than the macro-call that

it replaced. In general, however, expansion refers to the replacement of one sequence of lexemes by another during compilation. There are four kinds of expansion in BLISS:

- A lexical-function is replaced by its expansion, as described in *Section 15.4, "Lexical-Functions in General"* and *Section 15.5, "Specific Lexical-Functions"*.
- A lexical-conditional is replaced by its lexical-consequence or lexical-alternative, as described in *Section 15.6, "Lexical-Conditionals"*.
- A macro-call is replaced by the corresponding macro-body, and the formal-parameters in the macro-body are replaced by the corresponding actual-parameters, as described in *Section 16.2, "Macro-Declarations"* and *Section 16.3, "Macro-Calls"*.
- A require-declaration or library-declaration is replaced by the file it designates, as described in *Section 16.5, "Require-Declarations"* and *Section 16.6, "Library-Declarations"*.

Because the idea of replacing one entire sequence of lexemes with another, all at once, is inconsistent with the lexeme-by-lexeme processing described in *Section 15.1.2, "Lexeme-by-Lexeme Processing"*, the compiler processes a module in several stages; lexical processing is the first stage. The lexical processing stage of the compiler reads lexemes from the source file, collects lexemes until it can perform some lexical processing, passes the resulting lexemes to the next stage of the compiler, and, once again, reads lexemes from the source file.

The compiler can be thought of as working from a single sequence of lexemes, the *input stream*, as follows:

1. At the beginning of compilation, the input stream is the given module.
2. Each time the compiler can do nothing more without another lexeme, it takes a lexeme from the head of the input stream.
3. Whenever the compiler has accumulated a construct that can be expanded (such as a lexical-function or a macro-call), it processes that construct and places the resulting sequence of lexemes at the head of the input stream.
4. Whenever the lexical-processing stage of the compiler has accumulated a construct that cannot be further expanded (such as a keyword or a plus symbol), it passes that construct on to the next stage of the compiler.
5. When the input stream is empty, compilation is complete.

The method of lexical processing just described is simplified, but only in the following way: it suppresses those details of the BLISS compiler that, while they are important for efficient operation of the compiler, do not affect the meaning of the program or the object code produced by the compiler.

15.1.5. An Example of Lexical Processing

The following module provides an example of lexical processing:

```
MODULE S1 =
BEGIN
REQUIRE
    'STDMAC';
GLOBAL BIND
    P1 = STR8('ABC'),
    P2 = STR8('ABCDEFGHIJKLM');
```

```
END
ELUDOM
```

The fourth line of this module references the file named STDMAC. The contents of that file is assumed to be the following:

```
MACRO
    STR8(S) =
        %IF %CHARCOUNT(S) GTR 10
        %THEN %WARN('STR8 PARAM TOO LONG') %FI
        PLIT( %EXACTSTRING(10,%C' ',S) )
    %;
```

A detailed trace of the lexical processing of the module follows. The binding of names is described, expansions are performed, and the state of the compilation is given after each expansion.

The compiler starts with MODULE and reads lexemes from the input stream. The identifier S1 is treated in a special way because it is the module name; it does not affect the meaning of the program. When the compiler reads the semicolon on the fourth line, it knows that it has reached the end of a complete require-declaration. In accordance with the definition of require-declarations (*Section 16.5, "Require-Declarations"*), the compiler expands the require-declaration by placing the contents of the designated file at the head of the input stream.

At this point, the state of compilation is as follows:

```
    MODULE S1 =
    BEGIN
==> MACRO
        STR8(S) =
            %IF %CHARCOUNT(S) GTR 10
            %THEN %WARN('STR8 PARAM TOO LONG') %FI
            PLIT( %EXACTSTRING(10,%C' ',S) )
        %;
    GLOBAL BIND
        P1 = STR8('ABC'),
        P2 = STR8('ABCDEFGHIJKLM');
    END
    ELUDOM
```

The arrow at the beginning of the third line is a marker used in this explanation of lexical processing. Everything from the beginning of the module up to the arrow has passed through lexical processing and everything from the arrow through the end of the module is the input stream. The lexeme that immediately follows the arrow is the head of the stream.

The compiler continues processing lexemes, starting with MACRO. The occurrence of STR8 declares that name as a macro-name. The first occurrence of S declares that name to be the first (and only) formal parameter of STR8. The second and third occurrences of S are bound to this declaration. When the compiler reads the percent lexeme, it knows that it has read a complete macro-definition. It associates the macro-body with the name STR8.

The compiler continues, starting with GLOBAL. The occurrence of P1 declares that name to be a GLOBAL BIND name with a given value. The occurrence of STR8 is bound to the macro-declaration of the same name. When the compiler reads the right parenthesis that follows 'ABC', it knows that it has read a complete macro-call. In accordance with the definition of ordinary macros (*Section 16.2.3, "Semantics"*), the compiler expands the macro-call by placing a copy of the macro-body at the head of the input stream and replacing each formal-parameter in the copy by the corresponding actual-parameter.

At this point, the state of compilation is as follows:

```
MODULE S1 =
BEGIN
MACRO
    STR8(S) =
        %IF %CHARCOUNT(S) GTR 10
        %THEN %WARN('STR8 PARAM TOO LONG') %FI
        PLIT( %EXACTSTRING(10,%C' ',S) )
    %;
GLOBAL BIND
     P1 =
    ==> %IF %CHARCOUNT('ABC') GTR 10
        %THEN %WARN('STR8 PARAM TOO LONG') %FI
        PLIT( %EXACTSTRING(10,%C' ','ABC') ) ,
     P2 = STR8('ABCDEFGHIJKLM');
END
ELUDOM
```

The compiler continues, starting with the first lexeme, %IF, of the lexical-conditional. When the compiler reads the right parenthesis that immediately follows 'ABC', it knows that it has read a complete %CHARCOUNT lexical function. In accordance with the definition of that function (*Section 15.5.2, "String-Functions"*), the compiler expands the function by counting the number of characters in the actual-parameter 'ABC' and placing a numeric-literal that represents the count at the head of the input stream. Now the state of compilation is as follows:

```
...
GLOBAL BIND
    P1 =
        %IF ==> 3 GTR 10
        %THEN %WARN('STR8 PARAM TOO LONG') %FI
        PLIT( %EXACTSTRING(10,%C' ','ABC') ) ,
    P2 = STR8('ABCDEFGHIJKLM');
...
```

When the compiler reaches %THEN, it has evaluated the lexical-test of a lexical-conditional; because 3 is not greater than 10, the test is not satisfied. In accordance with the definition of lexical-conditionals (see *Section 15.6, "Lexical-Conditionals"*), the compiler skips the remainder of the lexical-conditional. The state of compilation is as follows:

```
...
GLOBAL BIND
    P1 = ==> PLIT( %EXACTSTRING(10,%C' ','ABC') ) ,
    P2 = STR8('ABCDEFGHIJKLM');
...
```

The compiler continues, starting with PLIT. The occurrence of %EXACTSTRING is recognized as a lexical-function name, and when the compiler reads the right parenthesis that follows, it knows it has a complete %EXACTSTRING lexical function. In accordance with the definition of that function (*Section 15.5.2, "String-Functions"*), the compiler makes 'ABC' into a 10-character quoted-string by filling at the right with blanks, and places this expansion at the head of the input stream.

The state of compilation is as follows:

```
...
GLOBAL BIND
    P1 = PLIT( ==> 'ABC       ' ),
```

```
    P2 = STR8('ABCDEFGHIJKLM');
...
```

The compiler continues, and reaches the declaration of P2. This declaration is treated similarly to that of P1; however, because the string given for P2 contains more than 10 characters, the test in the compilation-expression is satisfied and the compilation arrives at the following state:

```
...
GLOBAL BIND
    P1 = PLIT( 'ABC       ' ),
    P2 =
    ==> %WARN('STR8 PARAM TOO LONG') %FI
        PLIT( %EXACTSTRING(10,%C' ','ABCDEFGHIJKLM') );
...
```

The compiler expands the %WARN lexical-function by generating the warning message "STR8 PARAM TOO LONG", incrementing the warning count, and then placing the empty sequence (that is, nothing at all) at the head of the input stream. The compiler skips the %FI, which is the end of the lexical-conditional. Now the state of compilation is as follows:

```
...
GLOBAL BIND
    P1 = PLIT( 'ABC       ' ),
    P2 =
    ==> PLIT( %EXACTSTRING(10,%C' ','ABCDEFGHIJKLM') );
...
```

The compiler continues to the %EXACTSTRING lexical-function, which it expands as follows:

```
...
GLOBAL BIND
    P1 = PLIT( 'ABC       ' ),
    P2 = PLIT( 'ABCDEFGHIJ' ==> );
...
```

The compiler continues to the end of the input stream without performing any further binding or expansion. The result is the same as the result of compiling the following module:

```
MODULE S1 =
BEGIN
GLOBAL BIND
    P1 = PLIT( 'ABC       ' ),
    %WARN('STR8 PARAM TOO LONG')
    P2 = PLIT( 'ABCDEFGHIJ' );
END
ELUDOM
```

15.2. Quotation

BLISS has facilities for *quotation*. Quotation postpones until a later lexical scan the binding of a name and the expansion of a lexical-function or macro-call.

The need for quotation in BLISS is not obvious. The argument in favor of being able to quote a name is as follows:

- Some names are processed more than once. For example, a name in a macro-body is processed once as part of the macro-declaration and then, a second time, as part of the expansion of a macro-call.

- A particular use of a name can only be bound to one declaration. Therefore, a name that is processed twice could be bound in two different ways, and a choice must be made.
- A simple rule for choosing among bindings, such as "always bind a name the first time it is processed," is not flexible enough.

Therefore, some mechanism is necessary to specify when binding shall occur. This mechanism is the quotation facility.

The BLISS quotation facility has two parts: the quotation rules, and the quote-functions. Each quotation rule states that in a particular context certain kinds of names are bound or not bound. The quote-functions override the quotation rules and tell the compiler, for example, to quote a particular name regardless of the applicable quotation rules. The quotation rules are given later in this section.

A preliminary example of the BLISS quotation facility is as follows:

```
OWN
    X;
LITERAL
    MARK = 4;
MACRO
    M = MARK + %UNQUOTE MARK %;
...
BEGIN
LITERAL
    MARK = 5;
X = M;
...
END
```

The interesting part of the example is the binding of the uses of MARK. A detailed discussion follows.

The name MARK is declared twice, both times as LITERAL, but with different values. Each use of MARK must be bound to one or the other of these declarations.

The only uses of MARK are in the declaration of the macro M. There are two uses and they are handled in two different ways. The first occurrence is not bound because one of the BLISS quotation rules (defined in *Section 15.2.2, "Quotation Rules"*) states that in the macro-body of a macro-declaration only a macro formal-name is bound. The second occurrence is bound because the %UNQUOTE function (defined in *Section 15.5.14, "Quote-Functions"*) overrides the rule just stated and forces binding.

After processing, the macro-body is as follows:

```
MARK (not bound yet) + MARK (bound to LITERAL 4)
```

This macro-body is associated with the macro-name M.

Later in the processing of the example, the compiler replaces the macro-call on M with its expansion, and begins to process the expansion. This time around, the first MARK is bound because the quotation rules permit it. The second MARK is already bound and, because a name is never bound for a second time, is left as it is. After processing, the expansion is as follows:

```
MARK (bound to LITERAL 5) + MARK (bound to LITERAL 4)
```

Thus, the assignment statement is compiled as assigning 9 to X.

In this example, the application of the quotation rules to the binding of names has been illustrated. They also apply to the expansion of lexical-functions and macro-calls.

15.2.1. Quote Levels

The quotation rules of BLISS are organized around three *quote levels*. At any given time during compilation of a module, a particular quote level applies to the lexemes being read from the input stream. As compilation proceeds, the quote level changes depending on the language construct that is being compiled.

Quote levels are numbered from 1 to 3 and defined as follows:

1. *Normal-quote*. This level applies to any portion of a module not covered by the following quote levels.

2. *Name-quote*. This level applies to lexical contexts in which it is "natural" to ignore most applicable declarations. The portions of a module processed at name-quote level are as follows:

    a. A name that is about to be declared (explicitly or implicitly); specifically, a name that begins a definition within a declaration, or a name that appears in the formal-name-list of a routine, structure, or macro declaration.

    b. A name that appears in a name-quote actual-parameter of a lexical-function or any actual-parameter of a macro-call.

    c. An unreserved keyword in a context in which an unreserved keyword is required. An example is a module-switch in a module-head (described in *Section 18.1, "Psect-Declarations"*), where the context makes it clear that a keyword is being used as a switch. The BLISS keywords are listed in *Appendix A, "Predefined Identifiers"*.

3. *Macro-quote*. This level applies primarily to a macro-body in a macro-declaration. It also applies to a keyword-default-actual-parameter (*Section 16.2, "Macro-Declarations"*).

If more than one of the preceding levels could apply to a given context, the quote level with the highest number is chosen.

15.2.2. Quotation Rules

The quotation rules determine the binding of names and the expansion of both macro-calls and lexical functions. There are three quotation rules, one for each quote level, as follows:

1. At *normal-quote* level, bind every name.

    At this level, expand every macro-call and lexical-function.

2. At *name-quote* level, bind macro-names. That is, bind a name only if the binding, performed in the usual way, associates the name with a macro-declaration. At this level, expand every macro-call and lexical-function.

3. At *macro-quote* level, bind macro-formal-names. That is, bind a name only if the binding, performed in the usual way, associates the name with the (implicit) declaration of a macro-formal-name.

    At this level, expand only the quote lexical-functions: %QUOTE, %UNQUOTE, and %EXPAND.

The quote-functions, described in *Section 15.5.14, "Quote-Functions"*, are specifically designed to override the rules above. However, a quote-function only applies at a specific place in a program. For example, the %QUOTE function postpones application of the bind operation to a name that immediately follows the function, even though the quotation rules may call for binding of that name.

15.3. Lexical-Expressions

A module is presented to the compiler as a source file composed of characters and linemarks. During lexical processing, the characters are grouped into lexemes and then the lexemes are grouped into lexical-expressions.

A lexical-expression can be a single lexeme. Examples follow:

+	The plus symbol
MODULE	The keyword that begins a module
ALPHA	A name (not declared MACRO)
329	A decimal-literal
'ABC'	A quoted-string

Each of these examples is not only a single lexeme but is also *primitive*; that is, it is not expanded into some other sequence of lexemes during lexical processing.

Some examples of lexical-expressions that are more complicated are as follows:

%ASCIC'ABC'	A string-literal
%CHARCOUNT('ABC')	A lexical-function
%IF %SWITCHES(DEBUG) %THEN %WARN('BANG') %FI	A lexical-conditional with two nested lexical-functions
BETA(3,'ABC')	A macro-call (assume BETA is declared MACRO)
REQUIRE 'TBS';	A require-declaration
LIBRARY %STRING('XYZ',Q);	A library-declaration with a nested lexical-function

All of these lexical-expressions are composed of two or more lexemes. The first example is a %ASCIC string-literal and is primitive. The second example is a %CHARCOUNT lexical-function and is nonprimitive; it is expanded to 3, which is a primitive lexical-expression. The remaining examples are all nonprimitive, but their expansion requires contextual information not given here.

An example of a sequence of lexical-expressions that constitutes a complete module follows:

```
MODULE Q =
BEGIN
MACRO
    PACK(X) = UPLIT(%CHARCOUNT(X),X);
GLOBAL BIND
    MESSAGE = PACK('HELLO');
END
ELUDOM
```

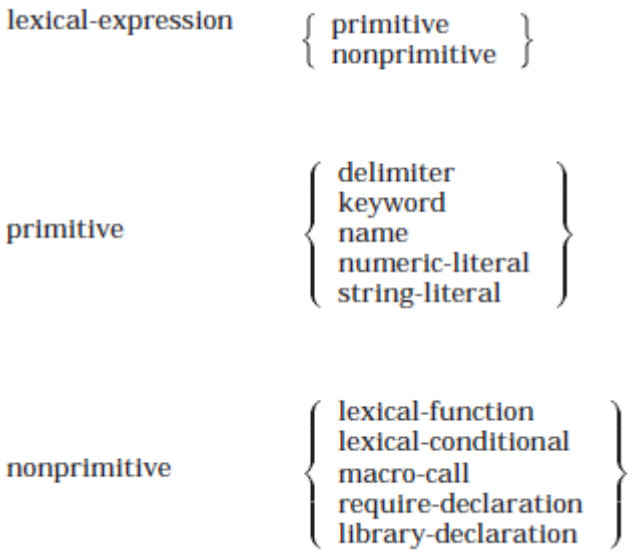
This module is mainly composed of primitive, single-lexeme lexical-expressions. The two exceptions are %CHARCOUNT(X) on the fourth line and PACK('HELLO') on the sixth line. The first nonprimitive lexical-expression, %CHARCOUNT(X), occurs within a macro-body and, therefore, is processed at macro-quote level; it is not expanded during macro definition, but is treated simply as a single-lexeme sequence. The PACK('HELLO') lexical-expression is a macro-call, and its expansion is as follows:

```
UPLIT(%CHARCOUNT('HELLO'),'HELLO')
```

This expansion includes the nonprimitive lexical-expression %CHARCOUNT('HELLO'). This is a lexical-function at normal-quote level, and its expansion is 5.

This section introduces the various kinds of lexical-expressions in BLISS and thus prepares for detailed descriptions in the remaining sections of this chapter.

15.3.1. Syntax



The primitive lexical-expressions are described in other parts of this manual; specifically, the *delimiters* are listed in *Section 2.2.1, "Lexemes"*, the *keywords* are listed in *Appendix A, "Predefined Identifiers"*, and the *names*, *numeric-literals*, and *string-literals* are described in *Chapter 4, "Primary Expressions"*.

Under certain conditions, a name, by itself, is also a macro-call; in that case, the name is nonprimitive.

15.3.2. Semantics

The fundamental lexical rule of BLISS is as follows:

```
A given sequence of lexemes is a valid BLISS module if and only if the
expansion of nonprimitive lexical-expressions produces a sequence of
 lexemes
that satisfies the definition of module given in Chapter 19.
```

This rule joins together the description of lexical-expressions given in this chapter and the definition of a module given in *Chapter 19, "Modules and Programs"*. That definition of a module includes, by reference, most of the other chapters of this manual.

The semantics of the various nonprimitive lexical-expressions are given in later sections of this chapter.

A few remarks about numeric- and string-literals as lexical-expressions are necessary. These remarks are presented here rather than in *Chapter 4, "Primary Expressions"* because they are closely related to the concepts of lexical processing.

15.3.2.1. Types of Numeric-Literals

The numeric-literals, as defined in *Section 4.2, "Numeric-Literals"*, can be classified as follows:

Fullword Type:	Unsigned Decimal-Literal Integer-Literal Character-Code-Literal
Single-Precision-Float Type:	Single-Precision-Float-Literal
Double-Precision-Float Type:	Double-Precision-Float-Literal

Different numeric-literals of the same type can be used interchangeably, but numeric-literals of different types cannot. For example, if a decimal-literal is called for in the syntax, then an integer-literal can be used instead, but a single-precision-float-literal cannot.

15.3.2.2. Types of String-Literals

The string-literals, as described in *Section 4.3, "String Literals"*, can be classified as follows:

Uncounted ASCII Type:	Quoted-String (without preceding string-type) %ASCII String-Literal %ASCIZ String-Literal
Counted ASCII Type:	%ASCIC String-Literal (BLISS–16/32 only)
Radix–50 Type:	%RAD50_11 String-Literal (BLISS–16/32 only) %RAD50_10 String-Literal (BLISS–36 only)
Sixbit Type:	%SIXBIT String-Literal (BLISS–36 only)
Packed Decimal Type:	%P String-Literal (BLISS–16/32 only)

Different string-literals of the same type can be used interchangeably, but string-literals of different types cannot. For example, if a quoted-string is called for, then a %ASCII string-literal can be used but a %ASCIC string-literal cannot.

BLISS permits this interchange of uncounted string-literals because each of them represents a sequence of ASCII characters. The zero at the end of a %ASCIZ literal is the ASCII "null" character, which has a 0 code.

The interchangeability of uncounted ASCII literals does make a slight addition to the language. Consider the definition of the %ASCIC string-literal (BLISS–16/32 only) given in *Section 4.3, "String Literals"*:

```
%ASCIC quoted-string
```

Because of the interchangeability of uncounted ASCII literals, the quoted-string can be replaced by an ASCIZ string-literal, and the result is as follows:

```
%ASCIC %ASCIZ quoted-string
```

Thus the following construct is a valid %ASCIC string-literal in BLISS–16 or BLISS–32:

```
%ASCIC %ASCIZ 'ABC'
```

This literal has a different interpretation from either %ASCIC'ABC' or %ASCIZ'ABC'. It is encoded in five bytes. The first byte contains the number of characters, 4, in the character sequence. The next three bytes contain the ASCII codes for A, B, and C. The final byte contains 0, which is the ASCII code for the null character.

Some further applications of interchangeability of uncounted ASCII literals follow:

```
%B %ASCII'11011'
%C %ASCII'Q'
%ASCII %ASCIZ %ASCII'ABC'
```

15.3.2.3. Numeric- and String-Literals

Except for the decimal-literal and quoted-string, the numeric- and string-literals are all composed of two lexemes. Each of these lexemes can be produced by nonprimitive lexical-expressions. An example is the following program fragment:

```
MACRO
    OCT(N) = %O %STRING(N) %;
...
OCT(23)
```

When the macro-call OCT(23) is expanded, the result is as follows:

```
%O %STRING(23)
```

Then the %STRING lexical-function is evaluated and the result is as follows:

```
%O '23'
```

Thus, the final value is 19 (decimal).

15.3.3. Discussion

Some nonprimitive lexical-expressions have an empty expansion; that is, they do not produce any lexemes. They are used for their side effects in controlling the compilation process. Two examples are the %UNQUOTE and %WARN lexical-functions described in *Section 15.2, "Quotation"*.

Other nonprimitive lexical-expressions have nonempty expansions, as do most of the lexical-expressions introduced so far. Almost all instances of this expanding type of nonprimitive lexical-expression can, in principle, be replaced by an equivalent sequence of primitive lexical-expressions. Such replaceable lexical-expressions do not produce any results that (again, theoretically) could not be obtained without them. Their purpose is to facilitate both conditional compilation and the writing of macros. Also, they often radically reduce the effort required to achieve a given result, and can be used to enhance the clarity of a module.

It is useful to examine those few cases in which a nonprimitive lexical-expression cannot in any way be replaced by an equivalent primitive lexical-expression sequence. There are three such cases. Each of them is rather specialized, and all of them involve lexical-functions. They are internal-only character sequences, excessively-long character sequences, and internal-only names.

An *internal-only character sequence* is a character sequence that is not composed entirely of printing characters, blanks, and tabs. Such character sequences can be represented by means of the %STRING and %CHAR lexical-functions, but cannot, according to *Section 4.3, "String Literals"*, be represented by a quoted-string.

As an example, consider the following character sequence:

A, carriage-return, line-feed, B

This sequence can be represented as follows:

```
%STRING('A',%CHAR(13),%CHAR(10),'B')
```

The lexical-functions %STRING and %CHAR are defined later, in *Section 15.5.2, "String-Functions"*. In this example, %CHAR(13) and %CHAR(10) represent the troublesome characters, and the %STRING function joins the four characters into a single sequence. That sequence cannot be represented by a quoted-string because a quoted-string cannot include a carriage return or a line feed. Thus, the uses of the %STRING and %CHAR functions are essential in this example.

An *excessively long character sequence* is one that contains more characters than can be represented on one line by a quoted-string. Such a character sequence can be represented on several lines by means of %STRING as follows:

```
%STRING('A line of many characters',
        'Another line of characters')
```

Again, %STRING is essential in this example.

An *internal-only name* is a character sequence that must be used as a BLISS name but that does not satisfy the syntax for a BLISS name. An example is XYZ.A, which is a valid assembler name but not a valid BLISS name. In BLISS, this name can be represented only as follows:

```
%NAME('XYZ.A')
```

The lexical-function %NAME is defined later, in *Section 15.5.4, "Name-Functions"*.

15.3.4. Pragmatics

The description of the lexical-processing stage of the compiler given in this chapter is correct with respect to the results of compilation, but does not reflect techniques that make the compiler itself more efficient. One such technique involves the use of internal encoding of lexemes, and another the use of multiple input streams for the expansion of lexical-expressions.

The latter technique merits some discussion, since it pertains to the scanning of lexemes. The compiler does not, in fact, maintain a single input stream into which the expansion of every lexical-expression is inserted. Instead, the compiler maintains several input streams. The principal input stream is the file for the module that is being compiled. However, a new input stream is introduced each time an expansion occurs. For example, after a macro-call has been processed, the corresponding macro-body becomes a new input stream. Even the replacement of a formal-name in a macro-body by the associated macro actual-parameter is done by treating the actual-parameter as a new input stream.

When a new input stream is introduced, input from the old input stream is suspended. Lexemes are taken from the new input stream until it terminates. This new stream can itself contain lexical-expressions whose expansion may introduce further new streams. When the end of an input stream is reached, the previous input stream is restored. Thus, the input streams are nested, and the initial input stream (the module file) is always the final input stream.

15.4. Lexical-Functions in General

A lexical-function is processed by the compiler. The result is a sequence of *lexemes* that is the expansion of the lexical-function. The expansion then becomes input to the compiler and is processed in its turn.

It is important to distinguish between the *evaluation* of a computational expression and the *expansion* of a lexical-function. A computational expression yields a value, and that value can be used in the evaluation of other expressions. In contrast, a lexical-function yields a sequence of lexemes, and that sequence can be used as input to the compiler.

It is also useful to distinguish between lexical-functions and macro-calls. Both return a sequence of lexemes, but a lexical-function invokes an operation that is built into BLISS, whereas a macro-call invokes an operation that must be defined in a macro-declaration. Thus, lexical-functions and macro-calls are related in the same way that executable-functions and routine-calls are related.

Certain parameters of lexical-functions can be expressions, but every such expression must be a compile-time constant expression. This restriction reflects the fact that all lexical-functions must be fully processed during compilation.

Each lexical-function begins with a keyword that, in turn, begins with a percent character; for example, %STRING and %CHAR.

A few examples of lexical-functions follow:

Lexical-Function	**Expansion**
%STRING('A','B','C')	'ABC'
%STRING('X',24)	'X24'
%CHARCOUNT('ABC')	3
%NUMBER('–00062')	–62 (written internally as one lexeme)

These are simple examples: the expansion of each of these lexical-functions is a single lexeme.

Some lexical-functions can return a sequence that is more than one lexeme in length. A simple example follows:

Lexical-Function	**Expansion**
%EXPLODE('ABC')	'A','B',C'

In this case, the expansion consists of five lexemes (three quoted-strings and two commas). Some lexical-functions are replaced by nothing (that is, an empty sequence of lexemes). For example, the following two lines produce the same object code:

```
Y = .A+%PRINT('CHECK POINT 20')F(X);

Y = .A+F(X);
```

However, the first version causes the informational message CHECK POINT 20 to be included in the output listing of the compiler.

Lexical functions can be nested. An example follows:

```
%STRING('A',%CHARCOUNT('XYZ'),'B')
```

Expansion of this %STRING function begins with the expansion of the nested %CHARCOUNT function as follows:

```
%STRING('A',3,'B')
```

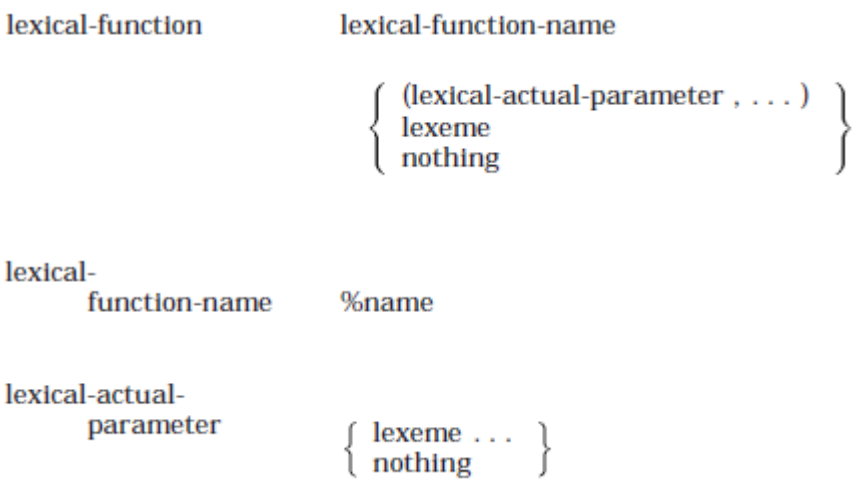
The %STRING function itself is then expanded as follows:

```
'A3B'
```

This quoted-string is the final expansion of the nested lexical functions.

This section gives the general definition of lexical functions, without defining any particular function. Specific definitions are given in the next section.

15.4.1. Syntax



15.4.2. Restrictions

A lexical-function must conform syntactically to one of the specific lexical-function definitions given in *Section 15.5, "Specific Lexical-Functions"*. For example, the %DECLARED function requires just one parenthesized parameter, and that parameter must be a single lexeme, specifically a name. Each lexical-function-name is a reserved keyword. It must not be declared and cannot be used for any other purpose.

15.4.3. Semantics

The processing of a lexical-function is performed as part of the compilation of a module. Processing begins when the compiler calls for the next lexeme of the input stream and that lexeme is recognized as a lexical-function-name. Processing continues until the last lexeme of a valid lexical-function has been processed. When processing is complete, the lexical-function is replaced by a sequence of lexemes that is its expansion.

You can prevent the processing of a lexical-function by placing a %QUOTE in front of it.

When processing of a lexical-function is complete and the lexical-function has been replaced by its expansion, the compiler takes its next lexeme from the beginning of the expansion. If the expansion is the empty sequence, the compiler takes its next lexeme from the stream that follows the lexical-function.

Most lexical-functions require a parenthesized list of actual-parameters. That parameter list can, itself, contain lexical-functions or macro-calls; it is no different in that respect than other portions of a BLISS module.

Each actual-parameter of a lexical-function is processed at either name-quote level or normal-quote level. For example, the first two actual-parameters of the %EXACTSTRING function are at normal-quote level, while the remaining actual-parameters are at name-quote level. In the individual definitions in *Section 15.5, "Specific Lexical-Functions"*, you indicate this distinction by placing a number sign (#) character before each parameter that is processed at name-quote level.

Once the actual-parameters have been processed, they must satisfy certain restrictions. The definition of each lexical-function gives restrictions that apply to its parameters. But one restriction applies to all

lexical-functions: when a parameter can be an expression, it must be a compile-time-constant-expression. This restriction is necessary because lexical-functions are always expanded during compilation.

A few lexical-functions cause the compiler to skip over a lexeme sequence that could otherwise be compiled. For example, %ERRORMACRO will, under certain circumstances, abort every macro-call expansion that is in progress. However, such lexical-functions never cause a portion of the unparsed input stream to be skipped; instead, they discard secondary sources of lexemes (macro-bodies) and proceed as if each of those macro-bodies had ended. Such lexical-functions are defined in *Section 15.5.12, "Advisory-Functions"* (%ERRORMACRO) and *Section 15.5.15, "Macro-Functions"* (%EXITITERATION and %EXITMACRO).

15.5. Specific Lexical-Functions

For purposes of this presentation, the lexical-functions are grouped as follows:

String-Functions	%STRING, %EXACTSTRING, %CHAR, %CHARCOUNT
Delimiter-Functions	%EXPLODE, %REMOVE
Name-Functions	%NAME, %QUOTENAME
Sequence-Test-Functions	%NULL, %IDENTICAL
Expression-Test-Functions	%ISSTRING, %CTCE, %LTCE
Bits-Functions	%NBITSU, %NBITS
Allocation-Functions	%ALLOCATION, %SIZE
Fieldexpand-Function	%FIELDEXPAND
Calculation-Functions	%ASSIGN, %NUMBER
Compiler-State-Functions	%DECLARED, %SWITCHES, %BLISS, %VARIANT
Advisory-Functions	%ERROR, %WARN, %INFORM, %PRINT, %MESSAGE, %ERRORMACRO
Titling-Functions	%TITLE, %SBTTL
Quote-Functions	%QUOTE, %UNQUOTE, %EXPAND
Macro-Functions	%REMAINING, %LENGTH, %COUNT, %EXITITERATION, %EXITMACRO
Require-Function	%REQUIRE

A description of these lexical-functions follows. The description begins with a brief discussion of quotation within lexical-functions. Then each class of lexical functions is described in its own section. Finally, all the lexical-functions are summarized in a single table.

15.5.1. Quote Levels for Lexical-Actual-Parameters

If a lexical-function appears in a context that is at macro-quote level, then the lexical-function is not expanded and its parameters are processed at macro-quote level. Otherwise, each parameter is processed at a quote level that is specified in the definition of the lexical-function.

In the definitions of lexical-functions that follow, a number sign (#) character sometimes appears before a parameter; in that case, the parameter is processed at name-quote level and is called a "name-quote parameter". Otherwise, the parameter is processed at normal-quote level.

For example, the definition of %EXACTSTRING in *Section 15.5.2, "String-Functions"* begins with the following:

%EXACTSTRING(n , fill , #p , . . .)

Therefore, the first two parameters of %EXACTSTRING are processed at normal-quote level and the remaining parameters are processed at name-quote level.

Note that the number sign character is part of the definition of BLISS; it never actually appears before a parameter in a program.

15.5.2. String-Functions

The string-functions operate on or produce quoted-string lexemes. They are important because they facilitate the compile-time manipulation of quoted-strings, and provide a useful basis for the definition of new macros. The string-functions also support the run-time functions for character handling that are described in *Chapter 20, "Character-Handling Functions"*.

Most of these functions convert a given sequence of lexemes into a different but essentially equivalent sequence of lexemes. The %STRING function converts a sequence of lexemes into a single quoted-string lexeme. The %EXACTSTRING function is like %STRING except that it adjusts the resulting quoted-string to a specified length. The %CHAR function takes a sequence of numeric values and converts it into a quoted-string lexeme.

The only string-function that does not perform a lexical conversion (as informally defined in the preceding paragraph) is %CHARCOUNT. This function forms a quoted-string and then yields a numeric-literal equal to the number of quoted-characters in the string.

The %STRING function plays a leading role among the lexical-functions because several lexical-functions are based on it. It accepts parameters that are each a quoted-string, numeric-literal, name, or empty sequence, and it puts these parameters together into a single quoted-string lexeme. For example:

Function	**Expansion**
%STRING('ABC','D')	'ABCD'
%STRING(23,%B'−111')	'23−7'
%STRING(ALPHA,,,9)	'ALPHA9'

The following lexical functions are all based on the %STRING function:

String-Functions	%EXACTSTRING, %CHARCOUNT
Delimiter-Function	%EXPLODE
Name-Functions	%NAME
Advisory-Functions	%ERROR, %WARN, %INFORM, %PRINT, %MESSAGE, %ERRORMACRO
Require-Function	%REQUIRE

Each of these lexical-functions begins by using the %STRING function to gather its parameters into a single quoted-string. Then the function performs an action on the quoted-string that is different for each function.

15.5.2.1. Definition

The string-functions are expanded as follows:

%STRING(#p , . . .)

Restriction: Each parameter must be one of the following:

- Fullword numeric-literal, that is:

  unsigned decimal-literal
  integer-literal
  character-code-literal

- ASCII string-literal, that is:

  quoted-string
  %ASCII string-literal
  %ASCIZ string-literal
  %ASCIC string-literal

- Identifier except for reserved keyword

- Empty sequence

Expansion: Modify each parameter, depending on what kind of lexeme it is, as follows:

- If the parameter is a *quoted-string*, then remove the initial and final quote characters.

- If the parameter is a *string-literal* with a string-type, then process the string-type (*Section 4.3, "String Literals"*), adding a leading or trailing character position as required, and remove the initial and final quote characters.

- If the parameter is a *numeric-literal*, then represent its value as a standard numeric-literal. A *standard numeric-literal* represents a positive value as

  a sequence of decimal digits that does not begin with 0, and represents a negative value as a minus sign followed by a sequence of digits that does not begin with 0.

- If the parameter is a *name*, change any lowercase letters to uppercase.

- If the parameter is an *empty sequence*, leave it as is.

Concatenate the modified parameters in the order given to form a single character sequence. Place the sequence in quotes, forming a quoted-string. Return the quoted-string.

%EXACTSTRING(n , fill , #p , . . .)
%EXACTSTRING(n , fill)

Restrictions: The parameter *n* must be a compile-time constant expression, and its value must satisfy implementation restrictions, given elsewhere, on the length of a character sequence.

The parameter *fill* must be a compile-time constant expression, and its value must be in the range 0 through 255. Use of a simple string-literal to represent a fill character is strongly discouraged because it will produce differing results in different dialects (see *Section 3.3, "Character Sequence Data"*). However, the character-code-literal (%C'character') is fully transportable.

Each of the remaining parameters must satisfy the restrictions on %STRING parameters.

Expansion: Evaluate the first two parameters. Then proceed as for the %STRING function, obtaining a single quoted-string from the third through last actual-parameters. If the function has only two parameters, form an empty quoted string ('').

Modify the resulting quoted-string as follows:

- If the quoted-string represents n characters, leave it unchanged.
- If the quoted-string represents more than n characters, remove quoted-characters from the right end until it represents n characters.
- If the quoted-string represents less than n characters, add quoted-characters at the right end until it represents n characters. Use the character whose ASCII code is given by the value of fill.

Return the resulting quoted-string.

%CHAR(code , . . .)

Restrictions: Each parameter must be a compile-time-constant-expression. The value of each parameter must be in the range 0 through 255.

Expansion: Evaluate each parameter and interpret its value as the code for an ASCII character. Concatenate the resulting characters to form a single character sequence. Return the quoted-string that represents that character sequence.

%CHARCOUNT(#p , . . .)

Restriction: The parameters must satisfy the restrictions on %STRING parameters.

Expansion: Proceed as for the %STRING function, obtaining a single quoted-string. Determine the number of quoted-characters (see *Section 4.3.1, "Syntax"*) in the quoted-string. Represent this number as a numeric-literal. Return the numeric-literal.

The result of a %STRING, %EXACTSTRING, or %CHAR function is a quoted-string. However, unlike the quoted-strings written by BLISS programmers, this quoted-string is not restricted to printing characters, blanks, and tabs; instead, it can represent any sequence of ASCII characters. This quoted-string is processed by the compiler as if it were an ordinary quoted-string.

15.5.2.2. Examples

The following are more illustrative than practical examples of string-function definitions:

Function	Expansion
%STRING('ABC')	'ABC'
%STRING('ABC','D')	'ABCD'
%STRING(%C'A')	'65'
%STRING('ABC',%C'A')	'ABC65'
%STRING(23)	'23'
%STRING(00023)	'23'
%STRING('00023')	'00023'

%STRING(20+3)	(INVALID: Operator not allowed)
%STRING('20+3')	'20+3'
%STRING(%B'–1111')	'–15'
%STRING(%O'77',%X'77')	'63119'
%STRING(%E'1.125E–02')	(INVALID: Float-literal not allowed)
%STRING(beta,'beta')	'BETAbeta'
%STRING(,X,,Y)	'XY'
%STRING(OWN,MODULE)	(INVALID: Reserved-keywords not allowed)
%STRING('OWN','MODULE')	'OWNMODULE'
%STRING(Q,18)	'Q18'
%STRING(Q,%DECIMAL–'18')	'Q–18'
%STRING(Q,–18)	(INVALID: Leading sign not allowed)

It is assumed in these examples that beta, X, Y, and Q are not macro-names. As %STRING parameters, non-macro names are treated literally (except for possible case conversion), whereas a macro-name is expanded.

In most situations, at least some of the parameters of the %STRING function (or any other lexical-function) are variable. For example:

```
%STRING(U,'=',V(X,Y))
```

Assume that U and V are declared as macros. The %STRING function will put the expansions of the two macros into a single quoted-string separated by an equal sign (=). If the expansions of U and V are 'ALPHA' and 'X+Y', respectively, then the final expansion of the %STRING function is the quoted-string 'ALPHA=X+Y'. Examples of the %EXACTSTRING function follow:

Function	**Expansion**
%EXACTSTRING(6,%C'X','ABC')	'ABCXXX'
%EXACTSTRING(3,%C'X','A BC')	'ABC'
%EXACTSTRING(2,%C'X', 'ABC')	'AB'
%EXACTSTRING(0,%C'X', 'ABC')	''
%EXACTSTRING(–2,%C'X','ABC')	(INVALID: Negative count)
%EXACTSTRING(4,%C'-')	'----'
%EXACTSTRING (6,%C'*',38,'–6')	'38–6**'
%EXACTSTRING(4,%C'Y', %C'X')	'88YY'
%EXACTSTRING(4,'Y','X')	'X' in BLISS–36 only
%EXACTSTRING(4,'Y','X')	'XYYY' in BLISS–16/32 only
%EXACTSTRING(4,%C'Y','X')	'XYYY' in all dialects
%EXACTSTRING(4,89,'X')	'XYYY'

Examples of the %CHAR function follow. They are assumed to lie in the scope of these declarations:

```
LITERAL
    ACODE = 65,
    BCODE = 66,
    APOSTROPHE = 39,
    CR = 13,
    LF = 10;
```

The examples follow:

Function	Expansion
%CHAR(65,66)	'AB'
%CHAR(ACODE,BCODE)	'AB'
%CHAR(ACODE+32)	'a'
%CHAR(ACODE,APOSTROPHE,BCODE)	'A"B' (3 characters)
%CHAR(CR,LF)	(new line)

Examples of the %CHARCOUNT function follow:

Function	Expansion
%CHARCOUNT('ABC')	3
%CHARCOUNT(,",)	0
%CHARCOUNT('A"C')	3

15.5.3. Delimiter-Functions

The delimiter-functions insert or delete delimiters within a given string. The %EXPLODE function forms a quoted-string and then "explodes" it into a list of single-character quoted-strings. It can be used to take a given string apart. The %REMOVE function deletes parentheses, brackets, or angle brackets that enclose a given actual-parameter.

15.5.3.1. Definition

The delimiter-functions are expanded as follows:

%EXPLODE(#p , . . .)

Restriction: Each parameter must satisfy the restriction on %STRING parameters.

Expansion: Proceed as for the %STRING function, obtaining a single quoted-string.

Remove the quotes from the ends of the resulting quoted-string, place each quoted-character in its own pair of quotes, and insert a comma between each quoted-string and the next.

Return the resulting sequence of quoted-strings and commas.

%REMOVE(#p)

Expansion: If the parameter begins and ends with a matched pair of parentheses, (. . .), brackets, [. . .], or angle brackets, < . . . >, then remove these lexemes from the parameter. Otherwise, leave the parameter unchanged.

Return the resulting sequence of lexemes.

The result of a %EXPLODE function is a sequence of one or more one-character quoted-strings. As with the %STRING, %EXACTSTRING, and %CHAR lexical-functions, these quoted-strings can represent any ASCII characters.

15.5.3.2. Examples

Examples of the %EXPLODE function follow:

Function	Expansion	
%EXPLODE('ABC')	'A','B', 'C'	(5 lexemes)
%EXPLODE('A')	'A'	(1 lexeme)
%EXPLODE( )	''	(1 lexeme)
%EXPLODE('A','B')	'A','B'	(3 lexemes)
%EXPLODE(%O'77')	'6','3'	(3 lexemes)
%EXPLODE('A',%O'–77')	'A','-','6','3'	(7 lexemes)

The following example is especially interesting:

```
%STRING(%EXPLODE('ABC'))
```

In this example, `%STRING` acts as the inverse of `%EXPLODE`, and the final expansion of the nested functions is just `'ABC'`.

Examples of the %REMOVE function follow:

Function	Expansion
%REMOVE((A,B,C))	A, B, C
%REMOVE(<A+1>)	A+1
%REMOVE([R(A+1)])	R(A+1)
%REMOVE((A+B))	A+B
%REMOVE((A)+(B))	(A)+(B)

This function is usually applied to macro-formal-names. A simple example of this application follows:

```
MACRO
     A(X) = RRR(%REMOVE(X))+1 %;
...
A(1);
A((1,2,3));
```

The extra parentheses in the second macro-call are required to keep its parameter from being treated as three parameters. The %REMOVE function deletes the extra parentheses, and the two macro-calls expand to the following:

```
RRR(1)+1;
RRR(1,2,3)+1;
```

Assuming that RRR is a conditional or iterative macro (as defined in *Section 16.3, "Macro-Calls"*) and thus accepts a parameter list of variable length, this is a useful result.

15.5.4. Name-Functions

Sometimes it is necessary to put together a name during program compilation. This need arises either because the name cannot be written in advance or because it is a sequence of characters that would not normally be accepted as a name.

15.5.4.1. Definition

The name-functions are expanded as follows:

%NAME(#p , . . .)
%QUOTENAME(#p , . . .)

Restriction: Each parameter must satisfy the restriction on %STRING parameters.

Expansion: Proceed as for the %STRING function, obtaining a single quoted-string.

Treat the sequence of quoted-characters in the quoted-string as a name. Return the resulting name.

The result of a %NAME and %QUOTENAME lexical-function is a name. Unlike user-defined names, this name is not restricted to the syntax for a BLISS name; instead, it can be any sequence of ASCII characters. It is accepted by the compiler as a name.

The %QUOTENAME lexical-function is similar to the %NAME function, the exception being that the resultant name is implicitly %QUOTED to prevent macro-expansion of the name.

15.5.4.2. Examples

The %NAME function permits the formation of a name at compile time. An example follows:

```
MACRO
    BLOCKOP(A) =
        OWN A: BLOCK[10];
        ROUTINE %NAME(A,'_INIT'): NOVALUE =
            BEGIN
            ...
            END;
        %;
```

Suppose this macro is called as follows:

```
macro is called as follows:
```

The expansion is as follows:

```
OWN BETA: BLOCK[10];
ROUTINE BETA_INIT: NOVALUE =
    BEGIN
    ...
    END;
```

The macro BLOCKOP uses the given name, BETA, for an OWN data segment. It uses %NAME to generate a related but distinct name, BETA_INIT, for the routine that initializes BETA.

The %NAME function also can be used to force the compiler to accept any character sequence as a name. That can be useful when something entirely new is needed. An example follows:

```
%NAME('+302')
```

Each time this construct appears, it is equivalent to writing just +302 and having those four characters accepted by the compiler as a valid name.

The %NAME function should not be used casually. Sometimes its use can cause an unexpected conflict with names generated by the compiler. For example, one compiler uses names like P.AAA, P.AAB, and so on, for PLIT storage. Furthermore, some operating systems restrict global names to characters that are in the RAD50 character set; in that situation, %NAME(+302) would be invalid as a global name.

The %NAME cannot be used to produce the "name" of a macro that is already declared; it will, however, always produce the macro expansion and may be used to invoke and expand a legitimately produced macro, as follows:

```
MACRO %NAME('A.B') = OWN X;%;
%NAME('A.B')                        !expands to "OWN X;"
```

There are also cases in which %NAME is essential. For example, the period character is used for global names in some software. Since period cannot be used in an ordinary BLISS name, %NAME must be used to form such a global name.

As an example of the use of the %QUOTENAME function, consider the following:

```
MACRO FOOBAR = .XYZ * 5 %;
...
UNDECLARE %NAME('FOO','BAR');
```

This would produce an error, because the compiler would interpret the UNDECLARE declaration as follows:

```
UNDECLARE .XYZ * 5
```

Moreover, inserting a %QUOTE before the %NAME, as follows, would again result in an incorrect compiler interpretation:

```
UNDECLARE %QUOTE %NAME('FOO','BAR')
```

However, using the %QUOTENAME function as follows:

```
UNDECLARE %QUOTENAME('FOO,'BAR')
```

results in a correct expansion to the following equivalent:

```
UNDECLARE %QUOTE FOOBAR;
```

15.5.5. Sequence-Test-Functions

A sequence-test-function expands to 1 or 0, depending on whether or not a certain condition is met. Because a test-function is expanded during compilation, it can be used within other lexical constructs. In particular, a sequence-test-function can be used as a compile-time-test in a lexical-conditional, as described in *Section 15.6, "Lexical-Conditionals"*.

The two test-functions, %NULL and %IDENTICAL, are applied to lexeme sequences. The %NULL function determines whether a sequence is empty, that is, contains nothing. The %IDENTICAL function compares two sequences to determine if they contain the same lexemes in the same order.

15.5.5.1. Definition

The sequence-test-functions are expanded as follows:

%NULL(#seq , . . .)

Expansion: Process the actual-parameters as for an ordinary macro-call, as defined in *Section 16.3.3.1, "Lexical Processing of Macro-Calls"*. Return the numeric-literal 1 or 0, depending on whether or not all the parameters expand to the empty sequence.

%IDENTICAL(#seq1 , #seq2)

Expansion: Process the actual-parameters, seq1 and seq2, as for an ordinary macro-call, as defined in *Section 16.3.3.1, "Lexical Processing of Macro-Calls"*. Return the numeric-literal 1 or 0, depending on whether or not the two resulting lexeme sequences are the same.

When two identifiers are compared, all letters are considered to be upper-case, so that case is effectively ignored. When two numeric-literals are compared, the numeric values of the numeric-literals are compared rather than the numeric-literals themselves.

15.5.5.2. Examples

Examples of the %NULL and %IDENTICAL functions follow:

Function	Expansion
%NULL( )	1
%NULL(,,)	1
%NULL(%EXACTSTRING(0,0,'ABC'))	0
%NULL(,ALPHA)	0
%IDENTICAL(A+B,A+B)	1
%IDENTICAL(,)	1
%IDENTICAL(3,%CHARCOUNT('ABC'))	1
%IDENTICAL(%O'77',63)	1
%IDENTICAL(ALPHA,alpha)	1
%IDENTICAL('ALPHA','alpha')	0
%IDENTICAL(A+B,A+C)	0
%IDENTICAL(32,'32')	0

The third example of %NULL is interesting, since it might be thought that a character sequence of length 0 would be a lexical sequence of length 0. However, the value of the following is the string-literal that represents the empty character sequence (") and that string-literal constitutes one lexeme:

```
%EXACTSTRING(0,0,'ABC')
```

15.5.6. Expression-Test-Functions

An expression-test-function expands to 1 or 0, depending on whether or not each of its parameters constitutes a particular form of expression. Since a test-function is expanded during compilation, it can be

used within other lexical constructs. In particular, an expression-test-function can be used as a compile-time-test in a lexical-conditional, as described in *Section 15.6, "Lexical-Conditionals"*.

The functions %ISSTRING, %CTCE, and %LTCE are applied to expressions. The %ISSTRING function determines whether or not each of its parameters is a string-literal. The %CTCE function determines whether or not each of its parameters is a compile-time-constant-expression. The %LTCE function determines whether or not each of its parameters is a link-time constant expression.

15.5.6.1. Definition

The expression-test-functions are expanded as follows:

%ISSTRING(exp , . . .)

Restriction: Each parameter must be a valid expression.

Expansion: Process each parameter, expanding all macro-calls and lexical-functions. Return the numeric-literal 1 if each of the resulting expressions is a quoted-string; return the numeric-literal 0 if any of the resulting expressions is not a quoted-string.

%CTCE(exp , . . .)

Restriction: Each parameter must be a valid expression.

Expansion: Process each parameter, expanding all macro-calls and lexical-functions. Return the numeric-literal 1 if each of the resulting expressions is a compile-time-constant-expression; return the numeric-literal 0 if any of the resulting expressions is not a compile-time constant expression.

%LTCE(exp , . . .)

Restriction: Each parameter must be a valid expression.

Expansion: Process each parameter, expanding all macro-calls and lexical-functions. Return the numeric-literal 1 if each of the resulting expressions is a link-time-constant-expression; return the numeric-literal 0 if any of the resulting expressions is not a link-time constant expression.

15.5.6.2. Examples

Examples of the expression-test-functions follow:

Function	Expansion
%ISSTRING('ALPHA', 'BETA','GAMMA')	1
%ISSTRING('ALPHA', 'BETA',GAMMA)	0
%ISSTRING(%ASCIC 'ALPHA')	1 (16/32 Only)
%ISSTRING(%RAD50_11'AB.99',%P'372')	1 (16/32 Only)
%ISSTRING(GET_STRING_RTN(BUF+I))	0
%ISSTRING(%CHARCOUNT('GAMMA'))	0
%ISSTRING(%STRING(%ASCIC'BETA'))	1
%ISSTRING('ABCDEFGHIJ')	1
%ISSTRING(PLIT('ABCDEFGHIJ'))	0

(Context for the following examples:

```
OWN X: REF VECTOR,
    Y: VECTOR[10];
EXTERNAL LITERAL A;
LITERAL V = 100; )
```

%CTCE (X,Y)	0
%CTCE (A)	0
%CTCE (V)	1
%CTCE (A,V)	0
%LTCE (X,Y)	1
%LTCE (X+A)	1
%LTCE (X[0])	0
%LTCE (Y[9])	1
%LTCE (V)	1

15.5.7. Bits-Functions

A bits-function determines the smallest number of bits required for the BLISS encoding of a given value. The %NBITSU function determines the number of bits required for an unsigned encoding, and the %NBITS function does the same for a signed encoding.

15.5.7.1. Definition

The bits-functions are expanded as follows:

%NBITSU(n , . . .)

Restriction: Each parameter must be a compile-time constant expression.

Expansion: This function calculates a bit count for each of its parameters. The bit count is the smallest number of bits required to represent the parameter as an unsigned binary integer. The following algorithm is used:

- If the function has just one parameter, evaluate that parameter.
  - If the value of the parameter is negative, then the desired bit count is %BPVAL (which, in BLISS–32 for example, is 32).
  - Otherwise, the desired bit count is the smallest integer, i, that satisfies the following relation:

    ```
    0≤vp≤(2**i)-1
    ```

    where *vp* is the value of the given parameter, and 2**i means "2 to the ith power".
- If the given %NBITSU function has several parameters, then the desired bit count is the value of the following expression:

  MAX(%NBITSU(*n1*), %NBITSU(*n2*), . . .)

  where *n1*, *n2*, and so on, are the given parameters.

Represent the bit count thus obtained as a numeric-literal. Return the numeric-literal.

%NBITS(*n* , . . .)

Restriction: Each parameter must be a compile-time constant expression.

Expansion: This function calculates a bit count for each of its parameters. The bit count is the smallest number of bits required to represent the parameter as a signed (two's complement) binary integer. The following algorithm is used:

- If the function has just one parameter, evaluate that parameter. The desired bit count is the smallest integer, i, that satisfies the following relation:

  ```
  -(2**(i-1))≤vp≤(2**(i-1))-1
  ```

  where *vp* is the value of the given parameter and 2**(i–1) means "2 to the (i–1)th power".

- If the given %NBITS function has several parameters, then the desired bit count is the value of the following expression:

  MAX(%NBITS(n1), %NBITS(n2), . . .)

  where n1, n2, and so on, are the given parameters.

  Represent the bit count thus obtained as a numeric-literal. Return the numeric-literal.

15.5.7.2. Examples

Examples of the %NBITSU and %NBITS functions follow:

Parameter List	Expansion of %NBITSU	Expansion of %NBITS
–8	%BPVAL	4
–1	%BPVAL	1
0	0	1
1	1	2
2	2	3
255	8	9
1,7	3	4
–8,7	%BPVAL	4
0,1,255,2,3	8	9

15.5.8. Allocation-Functions

An allocation-function determines the amount of storage required for a given kind of data. Allocation-functions are useful in laying out storage and calculating address offsets.

The %ALLOCATION function determines how many addressable units have been allocated for a given data name. The %SIZE function determines how many addressable units would be allocated for a given structure-attribute if that attribute were used in a data declaration.

15.5.8.1. Definition

The allocation-functions are expanded as follows:

%ALLOCATION(*name*)

Restriction: The parameter must be a name that is declared as one of the following:

OWN
GLOBAL
FORWARD
LOCAL
STACKLOCAL
REGISTER
GLOBAL REGISTER
EXTERNAL REGISTER

Expansion: Determine the number of addressable units allocated in the data segment for the given name. Represent the number just obtained as a numeric-literal. Return the numeric-literal.

%SIZE(structure-attribute)

Restriction: The parameter must be a structure-attribute, as described in *Chapter 11, "Data Structures"*.

Expansion: Determine the number of addressable units that would be allocated for a data structure if the given structure-attribute appeared in a data-declaration at this point in the program. A full description of structure-attributes is given in *Section 11.4, "Structure-Attributes and Storage Allocation"*. Represent the number just obtained as a numeric-literal. Return the numeric-literal.

15.5.8.2. Examples

The examples that follow are assumed to lie in the scope of these declarations:

```
GLOBAL
    X,
    Y: BYTE,            <= BLISS--16/32 only
    Z: VECTOR[10];
STRUCTURE
    ARRAY[I,J;M,N] =
        [M*N*4]
        (ARRAY+(I*N**)*4);
```

Examples of the %ALLOCATION and %SIZE functions follow:

Function	Expansion	Comment
%ALLOCATION(X)	%UPVAL	(For example, 1 in BLISS–36)
%ALLOCATION(Y)	1	(In BLISS–16/32 only)
%ALLOCATION(Z)	%UPVAL*10	(For example, 40 in BLISS–16)
%SIZE(VECTOR[10])	%UPVAL*10	(For example, 20 in BLISS–16)
%SIZE(VECTOR[10,WORD])	20	(In BLISS–16/32 only)
%SIZE(REF VECTOR)	%UPVAL	(For example, 1 in BLISS–36)

%SIZE(ARRAY[3,3])	%UPVAL*9	(For example, 36 in BLISS–32)

15.5.9. Fieldexpand-Function

The fieldexpand-function plays a specialized role in the declaration of data-structures. The function is used in conjunction with field-names, which are described in *Chapter 11, "Data Structures"*.

The %FIELDEXPAND function replaces a given field-name with its associated list of field-components. When an additional parameter is given, that parameter selects one of the field-components.

15.5.9.1. Definition

The field-functions are defined as follows:

%FIELDEXPAND(***field*** **)**
%FIELDEXPAND(***field*** **,** ***n*** **)**

Restrictions: The first parameter must be a field-name declared in a field-declaration.

The second parameter, if present, must be a compile-time constant expression, and its value, *v*, must lie in the range 0 through k–1, where k is the number of field-components associated with the field.

Expansion: Determine the list of field-components associated with the given field-name (see *Chapter 11, "Data Structures"*).

Represent each field-component as a standard numeric-literal (see the definition of %STRING); use a comma to separate each field-component in the list from the next.

If a second parameter is not given, return the entire list of field-components. Otherwise, return the vth field-component, where v is the value of the second parameter.

15.5.9.2. Examples

The examples that follow are assumed to lie in the scope of this declaration:

```
FIELD
    DCB_FIELDS =
        SET
        DCB_A = [0,0,0,0],
        DCB_B = [0,8,3,0],
        DCB_C = [0,11,5,1],
        DCB_D = [0,16,16,1],
        DCB_E = [1,0,%BPVAL,0]
        TES;
```

This declaration is taken from *Section 11.5, "Field-Declarations"*, where field-declarations are described and illustrated.

Examples of the %FIELDEXPAND function follow:

Function	**Expansion**	**Comment**
%FIELDEXPAND(DCB_A)	0,0,0,0	(7 lexemes)
%FIELDEXPAND(DCB_C)	0,11,5,1	(7 lexemes)

%FIELDEXPAND(DCB_C,0)	0	(1 lexeme)
%FIELDEXPAND(DCB_C,3)	1	(1 lexeme)

A field-name in a structure-reference is expanded without application of the %FIELDEXPAND function. Elsewhere, the %FIELDEXPAND function is necessary to force expansion.

15.5.10. Calculation-Functions

The calculation-functions provide a compile-time facility for calculating a value, saving it, and using it later in the compilation.

The %ASSIGN function assigns a value during program compilation. The value is obtained from a compile-time-constant-expression and is assigned to a COMPILETIME name. The %NUMBER function produces a numeric-literal from another numeric-literal, a quoted-string, or a name. When the %NUMBER function is applied to a name, the name must be a COMPILETIME, LITERAL, or GLOBAL LITERAL name.

15.5.10.1. Definition

The calculation-functions are expanded as follows:

%ASSIGN(#*name* , *n*)

Restrictions: The first parameter must be a name that is declared COMPILETIME.

The second parameter must be a compile-time-constant-expression.

Expansion: Evaluate the second parameter and associate the resulting value with the first parameter. Return the empty sequence.

%NUMBER(*p*)

Restrictions: The parameter must be a quoted-string, a numeric-literal, or a name.

If the parameter is a quoted-string, its quoted-characters must consist of an optional sign followed by a sequence of decimal digits. If the parameter is a numeric-literal, it must not be a float-literal. If the parameter is a name, it must be declared as one of the following:

LITERAL
GLOBAL LITERAL
COMPILETIME

Expansion: First determine the value of the parameter, as follows:

- If the parameter is a quoted-string, then remove the quotes and interpret the remainder as a decimal integer.
- If the parameter is a numeric-literal, use the value it represents.
- If the value is a name, use the value associated with the name by its declaration or, in the case of a COMPILETIME name, the most recently processed %ASSIGN function.

Once the value of the parameter has been determined, represent that value as a numeric-literal. Return the numeric-literal.

15.5.10.2. Example

An example of a macro that uses the %ASSIGN function appears in the following program fragment:

```
BEGIN
...
COMPILETIME
     ERRS = 0;
MACRO
     COUNT_ERROR = %ASSIGN(ERRS,ERRS+1) %;
...
END
```

The first declaration in this block declares ERRS as a COMPILETIME name. The second declaration declares COUNT_ERROR as a macro name. Wherever COUNT_ERROR is called, it will expand to the following:

```
%ASSIGN( ERRS, ERRS+1 )
```

Wherever the compiler encounters this expansion, it will increase ERRS by one. Thus the macro can be used to keep a count of a particular kind of error.

The combined use of the %ASSIGN and %NUMBER functions is the only way the value of a compile-time-constant-expression can be incorporated in a compile-time character sequence. An example follows:

```
COMPILETIME
    N = 0,
    Q = 4;
...
%ASSIGN(N,2*Q-1)
%INFORM('HERE IS AN INTEGER: ',%NUMBER(N))
```

The use of %ASSIGN is essential because 2*Q–1 is not a valid parameter for either %INFORM or %NUMBER.

More examples of the %NUMBER function follow. They are assumed to lie in the scope of the following declaration:

```
LITERAL
    Q = -16;
```

Examples are as follows:

Function	Expansion
%NUMBER('–180')	–180 (coded internally as one lexeme)
%NUMBER(83)	83
%NUMBER(%O'100')	64
%NUMBER(Q)	–16 (coded internally as one lexeme)

15.5.11. Compiler-State-Functions

Like the sequence-test-functions, a compiler-state-function expands to 0 or 1, depending on whether or not a certain condition is met. Since the function is expanded during compilation, it can be used within

other lexical constructs. In particular, a compiler-state-function can be used as a lexical-test in a lexical-conditional, described in *Section 15.6, "Lexical-Conditionals"*.

The compiler-state-functions refer to tables that are maintained by the compiler. The %DECLARED function determines whether a given name has been explicitly declared. The %SWITCHES function determines the settings of one or more compilation switches. The %BLISS function determines which compiler (BLISS– 16, BLISS–32, or BLISS–36) is in use. The %VARIANT function determines the integer value given in the /VARIANT qualifier switch (if any) in the compiler command line.

15.5.11.1. Definitions

The test-functions are expanded as follows:

%DECLARED(*#name*)

Restriction: The parameter must be a name.

Expansion: Return the numeric-literal 1 or 0, depending on whether or not it is explicitly declared at this point in the compilation of the program.

%SWITCHES(*#switch-name* , . . .)

Restriction: Each parameter must be one of the following on-off-switches:

ERRS | NOERRS
OPTIMIZE | NOOPTIMIZE
UNAMES | NOUNAMES
SAFE | NOSAFE
ZIP | NOZIP
CODE | NOCODE
DEBUG | NODEBUG

Expansion: Return the numeric-literal 1 or 0, depending on whether or not every parameter designates the current setting of an on-off-switch.

%BLISS(*#language-name*)

Restriction: The parameter must be one of the following compiler names:

BLISS16
BLISS32
BLISS36

Expansion: Return the numeric-literal 1 or 0, depending on whether or not the parameter designates the compiler that is compiling this program.

%VARIANT

Expansion: One of the following must apply:

- If the compiler command line contained a qualifier switch of the following form:

  /VARIANT:*n* or **/VARIANT=*n***

  where *n* is an unsigned decimal-literal, then return *n*.

- If the compiler command line contained a qualifier switch of the following form:

  /VARIANT

  then return the decimal-literal 1.

- If the compiler command line did not contain a **/VARIANT** qualifier switch, then return the decimal-literal 0.

15.5.11.2. Examples

The examples that follow are assumed to lie in the scope of only the following declarations:

```
OWN
    A,
    B;
SWITCHES
    OPTIMIZE,
    NOCODE;
UNDECLARE B;
```

It is further assumed that a BLISS–32 compiler is being used.

Examples of the %DECLARED, %SWITCHES, and %BLISS functions are as follows:

Function	Expansion
%DECLARED(A)	1
%DECLARED(B)	0
%DECLARED(C)	0
%SWITCHES(OPTIMIZE)	1
%SWITCHES(OPTIMIZE,NOCODE)	1
%SWITCHES(OPTIMIZE,CODE)	0
%BLISS(BLISS16)	0
%BLISS(BLISS32)	1
%BLISS(BLISS36)	0

15.5.12. Advisory-Functions

The advisory-functions generate compile-time output. The kind of advisory function determines the form of output: it may be an error message, a warning message, an informational message, or just a line in the program listing.

Two of the advisory functions do more than generate compile-time output: %ERRORMACRO also aborts any current macro-expansion, and %ERROR inhibits most subsequent expression evaluations and causes the object module to be discarded. See the appropriate BLISS user manual for further information on the side effects of %ERROR.

15.5.12.1. Definitions

The advisory-functions are expanded as follows:

%ERROR(#*p* , . . .)

Restriction: Parameters of an advisory-function must satisfy the restriction on parameters of the %STRING function.

Expansion: Proceed as for the %STRING function, obtaining a single quoted-string. Use the quoted-string as the text of a compiler error message, transmit the message as if it were a standard diagnostic, and add 1 to the compiler error count. Return the empty sequence.

%WARN(#*p* , . . .)

Restriction: Parameters of an advisory-function must satisfy the restriction on parameters of the %STRING function.

Expansion: Proceed as for the %STRING function, obtaining a single quoted-string. Use the quoted-string as the text of a compiler warning message, transmit the message as if it were a standard diagnostic, and add 1 to the compiler warning count. Return the empty sequence.

%INFORM(#, . . .)

Restriction: Parameters of an advisory-function must satisfy the restriction on parameters of the %STRING function.

Expansion: Proceed as for the %STRING function, obtaining a single quoted-string. Use the quoted-string as the text of a compiler information message, and transmit the message as if it were a standard diagnostic. Do not increment either the compiler error or warning count. Return the empty sequence.

%PRINT(#*p* , . . .)

Restriction: Parameters of an advisory-function must satisfy the restriction on parameters of the %STRING function.

Expansion: Proceed as for the %STRING function, obtaining a single quoted-string. Insert the character sequence directly into the compilation listing as the next line of that listing. Return the empty sequence.

%MESSAGE(#*p* , . . .)

Restriction: Parameters of an advisory-function must satisfy the restriction on parameters of the %STRING function.

Expansion: Proceed as for the %STRING function, obtaining a single quoted-string. Write the character sequence directly to the user's terminal (or other standard output device for the compilation). Return the empty sequence.

%ERRORMACRO(#*p* , . . .)

Restriction: Parameters of an advisory-function must satisfy the restriction on parameters of the %STRING function.

Expansion: Proceed as for the %STRING function, obtaining a single quoted-string. Use the quoted-string as the text of a compiler error message, transmit the message as if it were a standard diagnostic, and add 1 to the compiler error count. Then, in addition, abort every macro-call expansion that is currently in progress. Resume compilation of the program with the lexeme that follows the outermost of the aborted macro-calls.

15.5.12.2. Examples

Examples of the form of message produced by the advisory-functions appear in the BLISS user manual.

15.5.13. Titling-Functions

Each page of a compilation listing begins with a header. The header may vary from one implementation to another, but, typically, it includes the page number, compilation date, and other identifying information. By means of the titling-functions, a programmer can specify a *title* and a *subtitle* for inclusion in the header.

15.5.13.1. Definition

The titling-functions are expanded as follows:

%TITLE *qs*

Restriction: The lexeme qs must be a quoted-string. Note that qs is not enclosed in parentheses.

Expansion: Use the value of qs as the title in subsequent headers of the compilation listing. Return the empty sequence.

%SBTTL *qs*

Restriction: The lexeme qs must be a quoted-string. Note that qs is not enclosed in parentheses.

Expansion: Use the value of qs as the subtitle in subsequent headers of the compilation listing. Return the empty sequence.

These functions can be used repeatedly throughout a module, thus changing the title and/or subtitle from page to page of the listing.

15.5.13.2. Examples

Listing titles and subtitles appear in the BLISS user manual.

15.5.14. Quote-Functions

The quotation-functions are used to override the quotation rules given earlier, in *Section 15.2.2, "Quotation Rules"*. Each function applies to the name or lexical-function-name that immediately follows it. The %QUOTE function can also be applied to a comma or percent lexeme.

The %QUOTE function prevents a name from being bound and prevents expansion of a lexical-function or macro-call. The %UNQUOTE function causes a name to be bound but does not cause any expansion. The %EXPAND function causes both binding and expansion.

15.5.14.1. Definitions

The quote-functions are expanded as follows:

%QUOTE

Restrictions: The next lexeme must be a name, a lexical-function-name, a comma, or a percent sign.

Use of this function is restricted to macro-bodies or to the actual-parameters of a macro-call or lexical-function. That is, it applies only to lexemes encountered at macro-quote or name-quote level.

Expansion: Temporarily change the quotation rules so that binding of the next lexeme is deferred to a subsequent scan of the lexeme stream in which it occurs:

- If the next lexeme is an unbound name, an attempt to bind it will not occur when it is read.
- If the next lexeme is the beginning of a macro-call or lexical-function, an attempt to expand the macro-call or lexical-function will not occur when it is read.
- If the next lexeme is itself a quote-function in a macro-definition, that quote-function will be interpreted as a lexeme in the macro-body and thus will not, at that point, affect the binding of the lexeme which follows it.
- If the next lexeme is a comma in a list of actual-parameters in a lexical-function or macro-call, it will be interpreted as a lexeme in the current actual-parameter rather than as the separation between two actual-parameters.
- If the next lexeme is a percent in a macro-definition, it will be interpreted as a lexeme in the macro-body rather than as the termination of the macro-body.

Return the empty sequence.

%UNQUOTE

Restriction: The next lexeme must be a name or lexical-function-name.

Use of this function is restricted to macro-bodies or to the actual-parameters of a macro-call or lexical-function. That is, it applies only to lexemes encountered at macro-quote or name-quote level.

Expansion: Attempt to bind the next lexeme.

Forced binding of a macro-name or lexical-function-name does not also force expansion of the corresponding call or function.

Return the empty sequence.

%EXPAND

Restriction: The sequence of lexemes that follows %EXPAND must begin with a lexical-function or macro-call.

Use of this function is restricted to macro-bodies. That is, it applies only to lexemes encountered at macro-quote level.

Expansion: Temporarily change the quotation rules so that the lexical-function or macro-call that follows %EXPAND is expanded. Any macro-calls or lexical-functions contained in the expansion are not themselves automatically expanded.

Return the empty sequence.

15.5.14.2. Examples

A simple example of the use of the %UNQUOTE function is given earlier (in *Section 15.2, "Quotation"*). A series of more complex examples is given here. They are each based on the following program fragment:

```
MACRO
    Q1(P) = 1,P %,
    Q2 = 2 %,
    X = Q1(Q2) %;
...
ROUTINE R =
    BEGIN
    MACRO
        %QUOTE Q1(X) = 10,X %,
        %QUOTE Q2 = 20 %;
    BIND
        Y = UPLIT(%STRING(X));
    ...
    END;
```

When Q1(Q2) in the declaration of X is processed, neither Q1 nor Q2 is bound because they are names at macro-quote level (see *Section 15.2.1, "Quote Levels"*).

The %QUOTE functions are necessary in the second macro-declaration because Q1 and Q2 would otherwise be interpreted as macro-calls, and the declaration would become the following:

```
MACRO
    1,X = 10,X %,
    2 = 20 %;
```

which makes no sense. This expansion would occur because Q1 and Q2 are macro-names at name-quote level.

A call on the macro X appears in the bind-declaration. When X is expanded and processed, it is as follows:

```
10,20
```

This result reflects the fact that Q1 and Q2 are both bound in the scope of the second declarations of Q1 and Q2.

The following table shows the effect of using various quote-functions in the macro-body of the declaration of X:

If Q1(Q2) is replaced with:	**Then the processed expansion is:**
Q1(%UNQUOTE Q2)	10,2
%UNQUOTE Q1(Q2)	1,20
%UNQUOTE Q1(%UNQUOTE Q2)	1,2
%EXPAND Q1(Q2)	1,2
%EXPAND Q1(%QUOTE Q2)	1,20
Q1(%QUOTE Q2)	10,20
Q1(%QUOTE %QUOTE %QUOTE%QUOTE %QUOTE %QUOTE Q2)	10,Q2

The last two examples are especially interesting. In Q1(%QUOTE Q2), the %QUOTE has no effect because Q2 is at macro quote level and would not be bound or expanded anyhow.

In the final example, the many occurrences of %QUOTE have the effect of keeping Q2 from ever being expanded. The processed macro-body for this example is as follows:

```
Q1(%QUOTE %QUOTE %QUOTE Q2)
```

This macro-body becomes the expansion of X and must be processed as such; the result is as follows:

```
Q1(%QUOTE Q2)
```

Next, this macro-call is expanded. Before processing, the expansion is as follows:

```
10,%QUOTE Q2
```

Finally, this expansion is processed, giving the result shown, 10,Q2.

The preceding example is largely concerned with macro-names. That is not intended to imply that quote-functions are not important for lexical-functions or for names other than macro-names.

An example of %QUOTE applied to a comma and a percent sign is as follows:

```
MACRO
    X =
        MACRO
            Q(A) = UPLIT(A) %QUOTE %
        %;
X;
BIND
    Y = Q(4 %QUOTE, 5 %QUOTE, 6);
```

When the declaration of X is processed, the following macro-body is associated with X:

```
MACRO
    Q(A) = UPLIT(A) %;
```

The terminal percent gets into the macro-body because it was quoted in the declaration. The expansion of the macro-call X is exactly this same macro-body, and when it is processed, it establishes a declaration for Q.

The macro-call of Q has just one actual-parameter, as follows:

```
4, 5, 6
```

The commas get into the actual-parameter because they are quoted. The net effect of this example is to produce the following declaration:

```
BIND
    Y = UPLIT(4,5,6);
```

The following is an example of the use of %EXPAND:

```
MACRO
    B = C %,
    A = B %,
    X = A %,
    XX = %EXPAND A %;
UNDECLARE
    %QUOTE A,
    %QUOTE B;
OWN X;
```

```
OWN XX;
```

The macro-call X in the first OWN declaration is expanded to the name A with no further expansion, since the macro-name A has been undeclared.

The macro definition of XX is B, since the %EXPAND function forces expansion of the macro-call A within the macro-body for XX (prior to the undeclaration of macro-name A). Thus the macro-call XX in the second OWN declaration is expanded to B, again with no further expansion, since the macro-name B has been undeclared.

Note that the expansion of the function %EXPAND A within the macro-body for XX is not carried through to the name C. The following macro can be used to obtain this effect when desired:

```
MACRO
    FORCE [] = %QUOTE %EXPAND %REMAINING %;
```

The previous example could then be extended as follows:

```
MACRO
    B = C %,
    A = B %,
    X = A %,
    XX = %EXPAND A %,
    XXX = %EXPAND FORCE(A) %;
UNDECLARE
    %QUOTE A,
    %QUOTE B;
OWN X,
    XX,
    XXX;
```

The internally stored definition of FORCE is %EXPAND %REMAINING. When the macro-declaration of XXX is processed, the %EXPAND function causes the macro-call FORCE(A) to be expanded. Whenever a macro-call is expanded, all actual-parameters of the call are completely expanded. Therefore the actual-parameter A becomes C. That is, the body of FORCE expands simply to its fully expanded argument list.

The %EXPAND function has several practical applications:

- You can reduce compilation time by forcing a one-time expansion of embedded macro-calls at macro-declaration time, rather than at every occurrence of the containing macro-call.
- You can reduce the memory used during compilation for storing macro-bodies by forcing expansion of macros involving complicated conditional-compilation syntax.
- You can gain further efficiencies in the use of library files by forcing as much expansion as possible during the library pre-compilation.
- Macro-names declared for use within a library precompilation can be undeclared and thus freed for different uses in modules that refer to the library, if all instances of the macro-names are expanded within the library file.

15.5.15. Macro-Functions

The macro-functions are especially designed for use within macro-definitions; they are not useful in any other context. Complete definitions of the macro-functions are given in this section. However, these

definitions are difficult to understand without a discussion of macros. Examples and motivation for the macro-functions are given later, in *Section 16.3, "Macro-Calls"* on macro-calls and *Section 16.4, "Examples of Macros"* on examples of macros.

15.5.15.1. Definition

The macro-functions are expanded as follows:

%REMAINING

Expansion: Concatenate the actual-parameters not associated with formal-parameters, separating them by commas. Return the resulting sequence of lexemes.

%LENGTH

Expansion: Determine the number of actual-parameters for the current macro-call. Represent this number as a numeric-literal. Return the numeric-literal.

%COUNT

Expansion: Determine the recursion depth in a conditional-macro or the number of completed iterations in an iterative-macro. Represent this number as a numeric-literal. Return the numeric-literal.

%EXITITERATION

Expansion: Terminate the expansion of the current iteration of an iterative macro call. If a default separator or closing grouper (as specified in *Section 16.3.3.4, "Expansion of Iterative-Macros"*) is required at normal termination of an iteration, include it.

%EXITMACRO

Expansion: Terminate the expansion of the smallest macro-body in which this lexical-function is contained, just as if the terminal percent sign (%) lexeme appeared here.

15.5.15.2. Examples

Some examples of these functions are given as part of the discussion of macros in *Section 16.4, "Examples of Macros"*.

15.5.16. Require-Function

The require-function is the functional equivalent of the require-declaration (see *Section 16.5, "Require-Declarations"*); however, because it is not a declaration, %REQUIRE can appear in any context.

15.5.16.1. Definition

The require-function is defined as follows:

%REQUIRE(#P , . . .)

Restrictions: Parameters must satisfy the restrictions of the %STRING function (see *Section 15.5.2, "String-Functions"*).

The resulting quoted-string must be a valid file specification for the host operating system.

If the required file contains a %IF lexeme, it must also contain the matching

%THEN, %ELSE (if used), and %FI of the same lexical condition.

During the expansion of a required file (function or declaration), a fatal error will occur if the end of the file is found while a macro is still being declared.

A required file (function or declaration) must not appear during the expansion of a macro.

Expansion: Proceed as for the %STRING function, obtaining a single quoted-string for the required file. The specified file is then placed at the head of the input stream as the following actions are performed:

1. The file-name default rules for the host system and the compiler are applied.
2. Input from the current lexeme source is suspended.
3. The specified file is adopted as the current lexeme source.
4. Input from the suspended lexeme source is resumed when the specified file is empty.

15.5.16.2. Examples

The following depicts a required file named ADDMOD:

```
%IF %BLISS( BLISS32 )
%THEN
    ,ADDRESSING_MODE(
        EXTERNAL = LONG_RELATIVE)
%ELSE
    %IF %BLISS( BLISS16 )
    %THEN
        ,ADDRESSING_MODE(RELATIVE)
    %FI
%FI
```

And the following depicts how the file may be required:

```
MODULE A( %TITLE 'SETMODES' IDENT = '1-1'
       %REQUIRE('ADDMOD')
       ) =
BEGIN

...
END
ELUDOM
```

Note that unlike a require-declaration, the require-function can appear as a module-head-switch.

The following example shows a macro-declaration that produces a fatal error when called:

```
MACRO REQ = %REQUIRE('ERRMSG') %;
```

The error occurs because the %REQUIRE is encountered during the expansion of the macro.

The following example shows a macro-declaration that is allowed:

```
MACRO REQ = %EXPAND
            %REQUIRE('ERRMSG')
```

In the above example, the %EXPAND function expands the %REQUIRE function during the declaration of MACRO REQ. Notice that the percent lexeme, required for the termination of the macro-body, does not appear and is contained within the required file.

15.5.17. Summary of Lexical-Functions

The following table gives an example of each lexical-function:

Function	Expansion
%STRING('ABC',23,%B'−1111',,phi)	'ABC23−15PHI'
%EXACTSTRING(8,%C'X<single_quote>,'ABC',23)	'ABC23XXX'
%CHAR(65,66,67,39,97,98,99)	'ABC"abc'
%CHARCOUNT('ABC',23)	5
%EXPLODE('ABC',23)	'A','B','C','2','3'
%REMOVE(Q) [where Q is (A+1)]	A+1
%NAME('+302',beta)	+302BETA (as a name)
%QUOTENAME('FOO','BAR')	FOOBAR (as a quoted name)
%NULL('abc','')	0 (not a null sequence)
%IDENTICAL(ABC 5,ABC %B'101')	1 (sequences are identical)
%ISSTRING(BETA,'BETA')	0 (one not a string)
%CTCE(ALPHA[1])	0 (not a compile-time constant expression)
%LTCE(.ALPHA[1])	0 (not a link-time constant expression)
%NBITSU(7,2)	3
%NBITS(7,2)	4
%ALLOCATION(X) [scalar by default]	%UPVAL
%SIZE(VECTOR[10,WORD])	20 (BLISS−16/32 only)
%FIELDEXPAND(DCB_E)	1,0,%BPVAL,0
%ASSIGN(X,2+3) [X is COMPILETIME]	Empty (associates 5 with X)
%NUMBER(Y) [Y declared LITERAL 6]	6
%DECLARED(A)	1 (A is declared)
%SWITCHES(OPTIMIZE,NOCODE)	1 (these switches are on)
%BLISS(BLISS32)	1 (under BLISS−32 compiler)
%ERROR('error message')	Empty (steps error count)
%WARN('warning message')	Empty (steps warning count)
%INFORM('information message')	Empty
%PRINT('text in listing')	Empty
%MESSAGE('text for terminal')	Empty

%ERRORMACRO('error message')	Empty (aborts all macros)
%TITLE 'On Top Line of Page'	Empty
%SBTTL 'On Second Line of Page'	Empty
%QUOTE lexeme, comma, or percent	Empty
%UNQUOTE (Binds following name)	Empty
%EXPAND (Binds and expands)	Empty
%REMAINING	Unmatched actual-parameters
%LENGTH	Number of actual-parameters
%COUNT	Recursion or iteration count
%EXITITERATION	Empty (abort iteration)
%EXITMACRO	Empty (abort smallest macro)
%REQUIRE('ERRMSG')	Include specified file
%VARIANT	Return decimal-literal

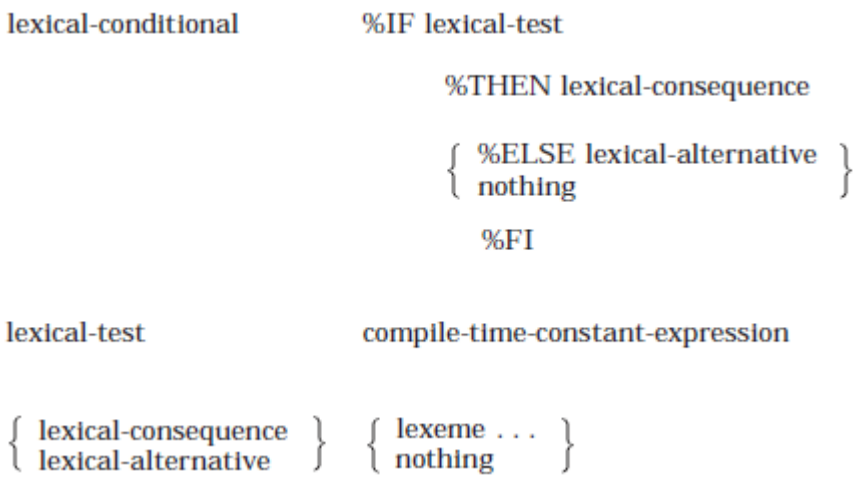
15.6. Lexical-Conditionals

A lexical-conditional evaluates a compile-time constant expression and then, depending on the value of that expression, skips one or the other of two given lexeme sequences. In some other programming languages, this kind of facility is called "conditional compilation".

Like the lexical-functions, a lexical-conditional is fully processed at compile-time. However, the lexical-conditional differs from a lexical-function in two respects. First, its syntax is different; that is just a matter of programming convenience. Second, and more important, it can be used to skip over a sequence of lexemes.

An example of a lexical-conditional is given in *Section 15.1.5, "An Example of Lexical Processing"*.

15.6.1. Syntax



The syntactic name *lexeme* is defined in *Section 2.2, "Lexemes and Spaces"*.

15.6.2. Restrictions

If a macro-body contains the lexeme %IF, then it must also contain the matching %THEN, %ELSE (if present), and %FI of the same lexical-conditional. This restriction must be satisfied by the source file before any lexical processing has been performed.

The restriction just given applies not only to a macro-body, but also to an actual-parameter in a macro-call or lexical-function, to the file that is designated by a require-declaration, or to the lexical-consequence or lexical-alternative within another lexical-conditional.

The keywords %IF, %THEN, %ELSE, or %FI must not be preceded by a quote-function.

15.6.3. Semantics

The expansion of a lexical-conditional begins with the evaluation of the lexical-test. If the low-order bit of the value of the lexical-test is 1, then the test is satisfied; otherwise, the test is not satisfied.

If the test is satisfied, the lexical-consequence is subjected to lexical processing and the lexical-alternative (if present) is skipped.

If the test is not satisfied, the lexical-consequence is skipped, and the lexical-alternative (if present) is subjected to lexical processing.

When a lexical-consequence or lexical-alternative is skipped, it is not processed in any way; the compiler scans through, looking for the terminating %ELSE or %FI and ignoring everything else.

A lexical-conditional in the macro-body of a macro-definition is not expanded; instead, it is included in the macro-body that is associated with the macro-name. Later, when the macro-body is used to expand a macro-call, the lexical-conditional is expanded.

15.7. Compile-Time Declarations

Compile-time variables provide a means to compute and assign values during compilation, particularly for use in combination with lexical-conditionals.

15.7.1. Syntax

compile-time-declaration	COMPILETIME compile-time-item , . . . ;
compile-time-item	compiletime-name = compile-time-value
compile-time-name	name
compile-time-value	compile-time-constant-expression

15.7.2. Semantics

The compile-time-declaration establishes a name whose value can be changed during compilation of the source module. In all other respects a compile-time-name is the same as a (non-GLOBAL) LITERAL name and can be used in all the same ways that a literal name can be used.

Observe that a compile-time-name must be given an initial value when the name is declared.

The value of a compile-time-name can be changed by the %ASSIGN lexical-function as described in *Section 15.5.10, "Calculation-Functions"*.

Chapter 16. Macros

Macros can make programs short and clear. When a certain construct is used often, a macro can be defined that gives the construct a name, and the name can then be used wherever the construct is required. By this means, a construct that is either large or unclear can be given a short, intuitive representation.

The idea of using the name of a construct instead of the construct itself can be extended in several ways, and BLISS has a variety of macro facilities. You can use simple macros in an obvious and intuitive way or you can use complicated macros to generate large and intricate tables.

This chapter discusses the macros and related facilities for user-defined expansion of source text. The first section introduces the various kinds of macros. The next two sections describe the declaration and call of macros. The final two sections describe the require- and library-declarations.

16.1. Introduction to Macros

The macro facilities of BLISS are important to learn because they can be used to add new notations to BLISS and thus greatly improve the organization and clarity of your programs. Other high-level programming languages that feature macro facilities generally provide limited capabilities; however, BLISS macro facilities are extremely innovative.

The expansion of macros is a part of lexical processing, and therefore macros are initially discussed at the beginning of *Chapter 15, "Lexical Functions"*. Specifically, the basic principles of macro expansion are presented in *Section 15.1.4, "Expansion"*, and an example is given in *Section 15.1.5, "An Example of Lexical Processing"*. An understanding of lexical processing is a prerequisite for the discussion of macros in this chapter.

This section provides an informal description of the basic *simple macro*, which introduces most of the general techniques of macro usage and is sufficient for most programming applications. If you do not have a strong interest in macros, you can read this section and skip the remaining descriptions.

16.1.1. Macro Declarations and Calls

A macro has two parts: the *macro-declaration* and the *macro-call*. A macro-declaration contains one or more *macro-definitions*, and each macro-definition associates a name, the *macro-name*, with a sequence of lexemes, the *macro-body*. Once a macro-name has been declared, it can be used in macro-calls.

An example of a macro-declaration follows:

```
MACRO
    CLA = PLIT(502,-1,3) %,
    ADD = PLIT(402,0,3) %;
```

This declaration contains two macro-definitions. The first macro-definition associates the name CLA with the macro-body PLIT(502,–1,3), and the second associates ADD with PLIT(402,0,3). Each macro-body is terminated by a percent sign (%) lexeme.

Two examples of macro-calls appear in the following example:

```
IF USED(REG)
    THEN CODE = CLA
```

```
    ELSE CODE = ADD;
```

The macro-calls here are CLA and ADD. If this conditional-expression is within the scope of the macro-declaration in the preceding paragraph, then it is equivalent to the following:

```
IF USED(REG)
    THEN CODE = PLIT(502,-1,3)
    ELSE CODE = PLIT(402,0,3);
```

Assuming that the names CLA and ADD have some mnemonic significance in the program from which this example is drawn, their use in the conditional-expression is certainly more clear than the use of the PLIT expressions.

A macro-body is processed twice. The first processing occurs when it is encountered as part of a macro-definition. During that processing, no object code is generated by the compiler; instead, the macro-body is saved by the compiler as a sequence of lexemes and that sequence is associated with the macro-name. The second processing occurs when the macro-body is used as the expansion of a macro-call. During that processing, the macro-body is compiled in the normal way.

16.1.2. Macros with Parameters

A macro-definition can have a list of formal-name parameters, and these formal-name parameters can appear in the macro-body. When a macro-call is expanded, each appearance of a formal-name parameter in the macro-body is replaced by the corresponding actual-parameter from the macro-call. The use of parameters in macros can greatly increase their power and generality.

An example of a macro with parameters follows:

```
MACRO
    GETBYTE(N,I) = ((N)^(-(I)) AND %B'11111111') %;
...
X = GETBYTE(.Y+1,12)-2;
```

In this example, the list of formal-names is (N,I) and the list of actual-parameters is (.Y+1,12). When the macro-call on GETBYTE is expanded, a copy of the macro-body associated with GETBYTE is made, and then N is replaced by .Y+1 and I is replaced by 12. The resulting expansion is as follows:

```
((.Y+1)^(-(12)) AND %B'11111111')
```

This expansion is placed at the head of the input stream (as described in *Section 15.1.4, "Expansion"* and is then compiled. Note that the expansion of GETBYTE(N,I) is an expression whose value is the 8-bit field (one byte) of N that is I bits from the right or low order end of N.

Actual-parameters are processed twice, as macro-bodies are. The first processing of an actual-parameter occurs when the macro-body is encountered as part of a macro-call. During that processing, no object-code is generated, just as for a macro-body. However, macro-calls, lexical-functions, or lexical-conditionals encountered within the actual-parameter are expanded during this first processing, and in this respect an actual-parameter differs from a macro-body. The second processing of the actual-parameter occurs during the expansion of a macro-call. During that processing, the actual-parameter is compiled like an ordinary sequence of lexemes.

16.1.3. Parenthesization of Macros

If a macro-body is an operator-expression, then it should be parenthesized; otherwise, a conflict of priority between the macro-body and its context may produce an unwanted interpretation. For the

same reason, each formal-name that is an operand of an operator-expression should be enclosed in parentheses.

The definition of GETBYTE, given above, follows the parenthesization guidelines just given. If it did not, that is, if the extra parentheses were not included, then the macro-declaration would be as follows:

```
MACRO
    GETBYTE(N,I) = N^(-I) AND %B'11111111' %;
```

and the assignment would become as follows:

```
X = .Y+1^(-12) AND %B'11111111'-2;
```

After insertion of default parentheses in accordance with operator priorities given in *Section 6.1.1, "Syntax"*, the assignment becomes the following:

```
X = (.Y)+(1^(-12)) AND (%B'11111111'-2);
```

This result is very different from that obtained previously, and the expression does not extract the desired byte value from N.

16.1.4. Quotation Rules and Macros

The quotation rules, described in *Section 15.2, "Quotation"*, have an important impact on macro usage. The following paragraphs present two examples of some of the less obvious effects of the quotation rules. The examples are concerned with the interpretation of constructs at the name-quote level.

Because the declaration of a name is at name-quote level, and because macros are expanded at that level, special measures are required to redeclare a macro-name. For example:

```
MACRO
    ALPHA = BETA %;
ROUTINE R =
    BEGIN
    LITERAL
        ALPHA = 1,
        %QUOTE ALPHA = 2;
    ...
    END
```

The first use of ALPHA in the LITERAL declaration is expanded before being declared, so that BETA is declared as a literal with value 1. The second use of ALPHA is quoted and therefore ALPHA is redeclared as a literal with value 2. Thus, within the routine R, BETA represents 1 and ALPHA represents 2.

Because a name in the formal-name list of a structure, routine, or macro-declaration is also at name-quote level, the consideration just illustrated applies to it.

Because an actual-parameter is processed at name-quote level, and because only macro-names are bound at that level, some unexpected results can occur. For example:

```
MACRO
    A(P1,P2) =
        BEGIN
        MACRO
            %QUOTE %QUOTE M = 1 %QUOTE %;
```

```
        LITERAL
            N = 2;
        OUTPUT(P1,P2);
        END %;
MACRO
    M = 10 %;
LITERAL
    N = 20;
...
A(M,N)
```

The macro-body for A is stored internally as follows:

```
BEGIN
MACRO
    %QUOTE M = 1 %;
LITERAL
    N = 2;
OUTPUT(P1,P2);
END
```

When the macro-call A(M,N) is expanded, its first actual-parameter, a macro-name, is bound and expanded but the second actual-parameter, a literal-name, is not bound (quotation rule 2). Thus the call is equivalent to the following:

```
A(10,N)
```

The expansion of this macro-call is as follows:

```
BEGIN
MACRO
    M = 1 %;
LITERAL
    N = 2;
OUTPUT(10,N);
END
```

Observe that the %QUOTE before the first occurrence of M prevents the replacement of that occurrence of M by 10, and thus keeps the macro-declaration valid. Observe, also, that the %QUOTE before the first % (percent) lexeme prevents the premature termination of the macro-body of A. The final result of lexical-processing is equivalent to the following:

```
OUTPUT(10,2)
```

Thus, N is finally bound to 2, not 20.

Although in most cases results will be as expected, the quotation rules emphasize that you must use macros carefully. Much of the need for quote-functions arises from the use of duplicate names within the scope of your program. However, because quote-functions add a level of complexity that can increase the chance of error, you should avoid such usage wherever possible.

16.1.5. A Survey of Macros and Related Facilities

The macros described in the preceding sections are *simple positional macros*; however, there are other kinds of macros. Moreover, BLISS has additional facilities that are not called macros, but are closely related to them. Macros and related facilities are discussed in this section.

BLISS has two main kinds of macros: *positional* and *keyword*. The difference between the two is in the way the actual-parameters of a macro-call are associated with the formal-names of the designated macro-declaration.

In a positional macro, the order of the actual-parameters is important; that is, the first actual-parameter is associated with the first formal-name, the second actual-parameter is associated with the second formal-name, and so on.

In a keyword macro, however, the order of the actual-parameters does not matter; instead, each actual-parameter is explicitly assigned to a formal-name. BLISS uses the word "keyword" in two ways. In classifying macros, the word designates a way of handling actual-parameters; elsewhere, it designates an identifier with a built-in meaning.

Positional macros are further classified as *simple*, *conditional*, and *iterative*. Simple macros are not only the simplest kind of macro but also the most commonly used. Conditional-macros and iterative-macros provide two ways of handling macros with a variable number of parameters.

The BLISS facilities that are related to macros are compiletime-declarations, require-declarations, library-declarations, and bound-declarations.

The *compile-time-declarations* are described in *Section 15.7, "Compile-Time Declarations"*. They are used to support macros. For example, a name that has been declared COMPILETIME can be used to designate a counter that is incremented each time a given macro is expanded.

The *require-declarations* are described in *Section 16.5, "Require-Declarations"*. Each require-declaration designates a file of BLISS declarations. When the require-declaration is processed, it is replaced by the designated file. A require-declaration can be viewed as a specialized form of macro that, in contrast to a true macro, can go to another file for its body.

The *library-declarations* are described in *Section 16.6, "Library-Declarations"*. A library-declaration is similar to a require-declaration except that it designates a file that has been preprocessed and thus requires minimal compilation. Library-declarations reduce compilation costs.

The *bound-declarations* are described in *Chapter 14, "Binding"*. They are used to associate a value with a name. Sometimes, you have a choice between a macro and a bound-declaration. In that situation, the bound-declaration is preferred. A bound-declaration not only makes the programmer's intentions more specific, but also is compiled more efficiently.

The BLISS macros and related facilities can be listed in outline form as follows: Macros and related facilities

- Macros
  - Positional macros
    - Simple macros

      Conditional macros

      Iterative macros
  - Keyword macros
- Related facilities

- Compile-time-declarations
- Require-declarations
- Library-declarations
- Bound-declarations

All of these facilities can be used to give a name to a programming construct and then to use that name instead of the construct. The construct can be an entire file of declarations as with a require-declaration, or a single integer, as with a literal-declaration. In any case, they can greatly improve the organization and clarity of a program.

16.2. Macro-Declarations

As the previous section states, every use of a macro has two parts: declaration and call. This section describes the macro-declarations for all kinds of BLISS macros.

A *positional-macro-declaration* consists of the reserved keyword MACRO, followed by a list of one or more macro-definitions. As with other declarations, the definitions are separated by commas and the declaration ends with a semicolon. Each macro-definition can be a simple-macro-definition, an iterative-macro-definition, or a conditional-macro-definition.

A *simple-macro-definition* consists primarily of a macro-name and a macro-body. The name is separated from the body by an equal sign, and the body is terminated by a percent sign (%) lexeme. The macro-name can, optionally, be followed by a parenthesized list of formal-names. The following macro-declaration contains a simple-macro-definition:

```
MACRO
SM1(F1,F2,F3) =
((F1(F2)+F1(F3))/2) %;
```

In this example, the name being declared is SM1, the formal-names are F1, F2, and F3, and the macro-body is as follows:

```
((F1(F2)+F1(F3))/2)
```

The percent sign lexeme after the macro-body is essential. Omission of the percent sign lexeme (a common programming error) causes the compiler to include in the macro-body everything it sees until it reaches either a subsequent percent lexeme or the end of the module.

A *conditional-macro-definition* is distinguished from a simple-macro-definition by an empty pair of square brackets inserted just before the equal sign. For example:

```
MACRO
CM1(F1,F2)[] =
((F1)^-(F2) + CM1(%REMAINING)) %;
```

In this example, the empty brackets ([]) identify the definition as a conditional-macro-definition.

An *iterative-macro-definition* is distinguished from a simple-macro-definition by an additional list of one or more formal-names that is enclosed in square brackets and inserted just before the equal sign. For example:

```
MACRO
IM1(F1)[F2] =
F1+F2 %;
```

In this example, the bracketed list of formal-names (just one, in this example), [F2], identifies the definition as an iterative-macro-definition.

A *keyword-macro-declaration* consists of the keyword KEYWORDMACRO followed by a list of one or more keyword-macro-definitions. A keyword-macro-definition is the same as a simple-macro-definition except that each formal-name can, optionally, have an explicit default-actual-parameter assigned to it. The default parameter is used when a call on the macro does not give the corresponding actual-parameter. For example:

```
KEYWORDMACRO
    COPYVECTOR(DEST,SOURCE,N=1) =
        INCR I FROM 1 TO N DO
            DEST[.I] = .SOURCE[.I] %;
```

In this example, the default-actual-parameter 1 is associated with the formal-name N. Defaults are not given for the other formal-names, DEST and SOURCE, so the empty lexeme sequence is the implicit default-actual-parameter for these formal-names. For this example, the macro-call must give actual-parameters for DEST and SOURCE, since the use of an empty lexeme sequence for either of these formal-names would yield an invalid macro expansion.

When a macro-definition is processed, the given macro-name is associated with the given macro-body. Aside from the recognition of formal-names within the macro-body, very little is done to the macro-body; it remains a lexeme sequence. No object code is generated during the processing of a macro-declaration.

In fact, the processing of a macro-declaration is a relatively small part of the processing of a macro. Only when macro-expansion is described, in *Section 16.3, "Macro-Calls"*, can motivation for different kinds of macro-declarations be provided.

16.2.1. Syntax

```
macro-declaration          { positional-macro-declaration }
                           { keyword-macro-declaration    }

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

positional-macro-
      definition           { simple-macro-definition      }
                           { conditional-macro-definition }
                           { iterative-macro-definition   }

simple-macro-
      definition           macro-name { (macro-formal-name , . . . ) } = macro-body %
                                      { nothing                      }

conditional-macro-
      definition           macro-name { (macro-formal-name , . . . ) } [ ]
                                      { nothing                      } = macro-body %

iterative-macro-
      definition           macro-name { (fixed-formal-name , . . . ) } [iterative-formal-name , . . . ]
                                      { nothing                      } = macro-body %

macro-name
macro-formal-name
fixed-formal-name
iterative-formal-name      name

macro-body                 { lexeme . . . }
                           { nothing      }

keyword-macro-
      declaration          KEYWORDMACRO
                                 keyword-macro-definition , . . . ;

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

keyword-pair               keyword-formal-name { = default-actual }
                                               { nothing        }

macro=name
keyword-formal-name        name
```

The syntactic name *lexeme* is defined in *Section 2.2, "Lexemes and Spaces"*.

16.2.2. Restrictions

Only a conditional-macro with one or more macro-formal-names can be used recursively. That is, the macro-body of any other macro must not contain a call on itself or a call on another macro that ultimately results in a call on the macro being defined.

A percent sign (%) in a macro-body must be quoted. It is quoted if it immediately follows an odd number of %QUOTE functions. For example:

```
%QUOTE

%QUOTE %QUOTE %QUOTE
```

Otherwise, the percent sign would terminate the macro-body.

A macro-body must not end with an odd number of %QUOTE functions. Otherwise, the the percent sign that terminates a macro-body would become part of the macro-body.

A default-actual in a keyword-macro-declaration must satisfy the restrictions on an actual-parameter in a macro-call. Literal commas must be quoted, parentheses must be balanced, and an odd number of quotes must not occur at the end; see *Section 16.3.2, "Restrictions"*.

16.2.3. Semantics

When the compiler encounters a macro-declaration, it processes the macro-definitions in the declaration one by one in the order in which they appear.

This section describes both the lexical processing and final interpretation of a macro-definition.

16.2.3.1. Lexical Processing of Macro-Definitions

Lexical processing of a macro-definition is performed at two quote levels, neither of which is the "normal" quote level. Indeed, the main reason BLISS has special quote levels is to properly support macro-definitions.

The following paragraphs specify the quote level for each part of a macro-definition. The definitions of the quote levels, given in *Section 15.2.1, "Quote Levels"*, are reviewed here.

The macro-body of a macro-definition is processed at macro-quote level. At this level, the compiler takes the following actions:

- Binds any occurrence of a name that is a formal-name in the macro-definition
- Expands any quote-function (%QUOTE, %UNQUOTE, or %EXPAND)

These actions are the minimum lexical processing. They leave most of the processing of a macro-body for later, when the macro is expanded at the point of call.

Each default-actual-parameter in a keyword-macro-definition is also processed at macro-quote level.

The macro-name and the formal-names (if any) are processed at name-quote level. At this level, the compiler takes the following actions:

- Binds macro-names only
- Expands lexical-functions and macro-calls

These actions can produce unexpected results, as illustrated in *Section 16.1.4, "Quotation Rules and Macros"*.

16.2.3.2. Interpretation of Macro-Definitions

As lexical-processing of a macro-definition is performed, the compiler forms the definition of a macro, which it retains for use when a call on the macro is encountered. The definition contains the following information:

- The kind of macro: simple, iterative, conditional, or keyword.
- The number of formal-names. For iterative macros, the distinction between fixed- and iterative-formal-names. For a keyword-macro, a list of the formal-names and the default-actual-parameters (if any) for each.
- A copy of the macro-body, with each formal-name properly identified as such.

16.2.4. Predeclared Macros

Three macro-names are predeclared in each BLISS dialect, %BLISS16, %BLISS32, and %BLISS36. The definition of these macro-names in BLISS–32, for example, is as follows:

```
MACRO
    %BLISS16[] = % ,
    %BLISS36[] = % ,
    %BLISS32[] = %REMAINING % ;
```

In each of the other dialects, the %REMAINING lexical function occurs in the definition of the appropriate name. This is not a valid declaration to give in a program because the names in the declaration begin with a percent sign (%) and are, in fact, reserved keywords rather than names (see *Appendix A, "Predefined Identifiers"*). However, the declaration does convey the interpretation given these identifiers.

The example declaration causes the BLISS–32 compiler to replace each call on %BLISS16 and %BLISS36 by the null lexeme and to replace each call on %BLISS32 by the actual-parameter sequence in the call. Each BLISS compiler predeclares these macro-names so that only the macro-name associated with the applicable language (BLISS–16, BLISS–32, or BLISS–36) expands to a non-null sequence.

By means of calls on these predeclared macros, you can specify processor dependencies. Then, when the program is compiled, only the actions relevant to the given processor are retained.

16.3. Macro-Calls

Once a macro has been defined, it can be invoked by a *macro-call*. BLISS has two kinds of macro-call, corresponding to the two main kinds of macro-declaration, positional and keyword. This section describes both kinds of macro-call.

A *positional-macro-call* consists of a macro-name followed by an optional list of actual-parameters. The list of parameters is normally enclosed in parentheses; however, square brackets or angle brackets can

be used instead, without changing the interpretation of the call. An actual-parameter can be nearly any sequence of lexemes.

An example of a positional-macro-call follows:

```
ALPHA(A,.B+3,'qrs' 16 MODULE)
```

In this example, the macro-name is ALPHA. The first and second actual-parameters are A and .B+3, which happen to be valid BLISS expressions; however, they are not compiled as such until after the call has been expanded. The third actual-parameter is a sequence of three lexemes that does not appear to make sense in BLISS; however, there is nothing inherently wrong with the use of this sequence as a macro actual-parameter. In order for this example to be a valid macro-call, it must lie within the scope of a declaration of ALPHA as a positional macro; and that declaration must make some valid use of the given actual-parameters.

A *keyword-macro-call* is similar to a positional-macro-call except that a name must be associated with each actual-parameter. The name and actual-parameter are separated by an equal sign. The name must be one of the keyword-formal-names in the definition of the given macro.

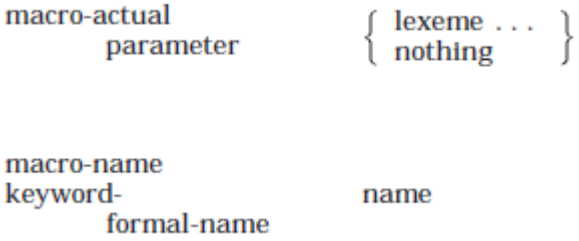
An example of a keyword-macro-call is as follows:

```
GAMMA(X=Q(R,1),Y=3)
```

It is assumed that this call occurs in the scope of a declaration of GAMMA as a keyword-macro name. That declaration must have X and Y as formal-names.

16.3.1. Syntax

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

positional-
        macro-call
        macro-name { ( macro-actuals ) }
                   { [ macro-actuals ] }
                   { < macro-actuals > }
                   { nothing           }

macro-actuals
        { macro-actual-parameter , . . . }
        { nothing                        }

keyword-
        macro-call
        macro-name { ( keyword-assignments ) }
                   { [ keyword-assignments ] }
                   { < keyword-assignments > }

keyword-
        assignments
        { keyword-assignment , . . . }
        { nothing                    }

keyword-
        assignment
        keyword-formal-name = macro-actual-parameter
```



The syntactic name *lexeme* is defined in *Section 2.2, "Lexemes and Spaces"*.

16.3.2. Restrictions

The macro-name in a positional-macro-call must be declared in a positional-macro-declaration. Similarly, the macro-name in a keyword-macro-call must be declared in a keyword-macro-declaration.

Each keyword-assignment in a keyword-macro-call must begin with a formal-name from the declaration of the designated keyword-macro. No formal-name can be used more than once in a keyword-macro-call.

A macro-actual-parameter must not contain unbalanced parentheses or brackets (<> or []). That is, every left parenthesis must be followed (somewhere in the same macro-actual-parameter) by a matching right parenthesis; every left square bracket, by a matching right square bracket; and every left angle bracket by a matching right angle bracket.

A comma (,) in a macro-actual-parameter must be quoted or parenthesized. It is quoted if it immediately follows an odd number of %QUOTE functions. It is parenthesized if it is enclosed in a balanced pair of parentheses or brackets that is, itself, contained in the macro-actual-parameter.

A macro-actual-parameter must not end with an odd number of %QUOTE functions. Otherwise, the following comma would be quoted.

If the macro-name of a macro-call is declared as a simple macro with no formal-names, then the macro-call must consist of just the macro-name. This does not mean that the macro-name cannot be followed by something that looks like a parenthesized list of actuals; it only means that the compiler will not process that construct as part of the macro-call.

If the macro-name of a macro-call is declared other than as a simple macro-call with no formal-names, then the macro-call must have a parenthesized (or bracketed) list of actual-parameters. The list can be empty, but the pair of parentheses or brackets must be there.

16.3.3. Semantics

A macro-call is first subjected to lexical-processing and then expanded. Lexical-processing is the same for all macro-calls, and is described in the next section. Expansion is different for the different kinds of macros, and is described in four separate sections.

The expansion of a macro-call can be cut short by a %EXITITERATION or %EXITMACRO lexical-function; these functions are described in *Section 15.5.15, "Macro-Functions"*.

16.3.3.1. Lexical Processing of Macro-Calls

The processing of a macro-call begins when a macro-name is bound to a macro-declaration.

Once a macro-name has been bound, the actual-parameters (if any) are processed at name-quote level. At this level, the compiler takes the following actions:

- Binds macro-names only
- Expands lexical-functions and macro-calls

Because the compiler expands lexical-functions and macro-calls at this level, an expansion can occur within another expansion. The actual-parameters of a macro-call are separated by commas. However, a comma that is quoted or parenthesized is treated literally. See *Section 16.3.2, "Restrictions"* for the definition of a quoted or parenthesized comma.

The list of actual-parameters is terminated by the right parenthesis or bracket that matches the left parenthesis or bracket that begins the list.

The following list gives some macro-calls and identifies the actual-parameters in these calls. Because some macro-calls are included in the actual-parameters, the following macro-definitions are given first:

```
MACRO
    M1(F1,F2) = F1, F1/F2, F1*F2 %,
    M2        = A, B, C, D %;
```

The identification of the actual-parameters *a1*, *a2*, . . . is given in the following list:

Macro-Call	**a1**	**a2**	**a3**	**a4**
M3(X,Y,Z)	X	Y	Z	
M3(X,Y %QUOTE, Z,W)	X	Y,Z	W	
M3(1(X,Y))	X	X/Y	X*Y	
M3(M2)	A	B	C	D
M3(X,%QUOTE M1(X,Y),Z)	X	M1(X,Y)	Z	
M3(X,(Y,Z),W)	X	(Y,Z)	W	
M3(X,F[M2],Y)	X	F[A,B,C,D]	Y	

16.3.3.2. Expansion of Simple Macros

The compiler uses the following algorithm for expanding a simple macro-call:

1. **Associate actuals with formals –** Associate the first actual-parameter with the first formal-name of the corresponding definition, the second actual-parameter with the second formal-name, and so on.
    a. If there are too many actual-parameters, save the extra actual-parameters for use in the value of %REMAINING.
    b. If there are too few actual-parameters, associate the empty lexeme sequence with each formal-name that does not have an actual-parameter.
2. **Prepare macro-body –** Make a copy of the macro-body of the corresponding definition. In the copy, replace each unquoted occurrence of a formal-name with the corresponding actual-parameter.
3. **Expand macro-functions –** Replace certain lexical-functions in the copy of the macro-body as follows:

a. %LENGTH becomes an unsigned integer-literal that represents the number of parameters in the list of actual-parameters.

b. %REMAINING becomes a list of the extra actual-parameters.

   If the macro-definition has n formal-names, then %REMAINING is replaced by the following lexeme sequence: the (n+1)th actual-parameter, a comma, the (n+2)th actual-parameter, a comma, and so on, ending with the last actual-parameter.

   If there are no extra actual-parameters, %REMAINING is replaced by the empty lexeme sequence.

c. %COUNT becomes zero.

4. **Place expansion in stream –** Place the modified copy of the macro-body at the head of the input stream.

16.3.3.3. Expansion of Conditional Macros

The compiler uses the following algorithm for expanding a conditional macro-call:

The semantics of conditional-macros is quite similar to those of simple-macros. In the following, each item that differs from simple-macros is marked with an asterisk (*).

1. **Associate actuals with formals –** Associate the first actual-parameter with the first formal-name of the corresponding definition, the second actual-parameter with the second formal-name, and so on.

   a. If there are too many actual-parameters, save the extra actual-parameters for use in the value of %REMAINING.

   b. *If there are too few actual-parameters, use the empty lexeme sequence as the expansion of the macro-call and exit from this algorithm.

   c. *If there are no actual-parameters in the call and no formal-names in the macro-definition, use the empty lexeme sequence as the expansion of the macro-call and exit from this algorithm.

2. **Prepare macro-body –** Make a copy of the macro-body of the corresponding definition. In the copy, replace each unquoted occurrence of a formal-name with the corresponding actual-parameter.

3. **Expand macro-functions –** Replace certain lexical-functions in the copy as follows:

   a. %LENGTH becomes an unsigned integer-literal that represents the number of parameters in the list of actual-parameters.

   b. %REMAINING becomes a list of the extra actual-parameters.

      If the macro-definition has n formal-names, then %REMAINING is replaced by the following lexeme sequence: the (n+1)th actual-parameter, a comma, the (n+2)th actual-parameter, a comma, and so on, ending with the last actual-parameter.

      If there are no extra actual-parameters, %REMAINING becomes the empty lexeme sequence.

   c. *%COUNT becomes an unsigned integer-literal that represents the depth of recursion for this macro.

If the macro-definition has no formal-names, then recursion is not permitted, and %COUNT always becomes 0.

The *depth of recursion* is the number of calls on the same macro that occurred prior to the current call and are still in the process of being expanded.

4. **Place expansion in stream –** Place the modified copy of the macro-body at the head of the input stream.

16.3.3.4. Expansion of Iterative-Macros

The compiler uses the following algorithm for expanding an iterative macro-call:

1. **Associate actuals with fixed-formals –** Associate the first actual-parameter with the first fixed-formal-name of the macro-definition, associate the second actual-parameter with the second fixed-formal-name, and so on.

   a. If there are one or more extra actual-parameters, call them the *remaining-actuals-list*, and go to step 2.

   b. Otherwise, use the empty lexeme sequence as the expansion of the macro-call and exit from this algorithm.

2. **Prepare fixed-macro-body –** Make a copy of the macro-body of the designated macro-definition. In that copy, replace each unquoted occurrence of a fixed-formal-name by the corresponding actual-parameter. Call the result the *fixed-macro-body*.

3. **Expand %LENGTH macro-function –** Replace any %LENGTH lexical-function in the macro-body with its expansion, as follows:

   a. %LENGTH becomes an unsigned integer-literal that represents the number of parameters in the list of actual-parameters.

The next four steps, step 4 through step 7, are a loop. Each pass through the loop generates a new copy of the macro-body. These copies are placed on the input stream in step 8.

4. **Associate actuals with iterative-formals –** Associate the first actual-parameter of the remaining-actuals-list with the first iterative-formal-name of the macro-definition, associate the second actual-parameter with the second iterative-formal-name, and so on.

   As each actual-parameter is associated with an iterative-formal-name, remove it from the remaining-actuals-list. If there are too few actual-parameters, associate the empty lexeme sequence with each iterative-formal-name that does not have an actual-parameter.

   Steps 1a and 7 of this algorithm guarantee that there will always be at least one remaining actual-parameter at the beginning of this step.

5. **Prepare iterative-macro-bodies –** Make a copy of the fixed-macro-body (obtained in steps 2 and 3). In that copy, replace each unquoted occurrence of an iterative-formal-name by its associated actual-parameter (obtained in step 4).

6. **Expand other functions –** Replace any occurrences of the %COUNT or %REMAINING function in the iterative-macro-body as follows:

a. %COUNT becomes an unsigned numeric-integer that represents the iteration count for this iteration.

   The *iteration count* is the number of completed iterations; thus the count is 0 the first time this step is executed, 1 the second time, and so on.

b. %REMAINING becomes the remaining-actuals-list.

7. **End test –** If the remaining-actuals-list is not empty, go back to step 4.

8. **Place expansion in stream –** Place the following sequence of lexemes at the head of the input stream:

   a. The default left grouper, if any.

   b. The copies of the macro-body prepared in step 4 through step 6. Place a default separator between each pair of copies.

   c. The default right grouper, if any.

The final step of the algorithm just given requires *default punctuation*. Specifically, step 8b requires a default separator, and step 8a and step 8c require default groupers.

The selection of default punctuation for a given macro-call depends on the one or two lexemes that immediately precede the macro-call. Those lexemes are called the *left context*, and they are examined only after their lexical processing is complete.

BLISS has five combinations of default separator and default groupers. The first three use a comma, a semicolon, or an operator as the separator and do not use groupers. The fourth uses a semicolon as a separator and parentheses as groupers. The fifth uses a semicolon as a separator and SET and TES as groupers.

The left context for each of the five combinations is given in the following list, together with remarks that show why those defaults are appropriate.

1. **Comma separators, no groupers –** In the following cases, the default separator is a comma and default groupers are not used:

Left Context	**Remarks**
([<	The expansion serves as a list of actual-parameters, formal-names, or plit-items.
The keyword phrase at the beginning of a declaration.	The expansion serves as a list of declaration-items.
, (comma)	The expansion serves as the continuation of a list of actual-parameters, formal-names, plit-items, or declaration-items.

   This case does not apply to a left parenthesis that is the first lexeme of a block or an expression.

2. **Semicolon separators, no groupers –** In the following cases, the default separator is a semicolon and default groupers are not used:

Left Context	Remarks
BEGIN (	The expansion serves as the contents of a block as defined in *Section 8.1.1, "Syntax"*.
SET	The expansion serves as a sequence of case-lines in a case-expression or select-lines in a select-expression.
Leading keyword of control-expression	Not a useful default.
CODECOMMENT	Not a useful default.
; (semicolon)	The expansion serves as the continuation of a sequence of declarations, block-actions, case-lines, or select-lines.

This case applies to a left parenthesis "(" only if it is the first lexeme of a block or an expression.

3. **Operator separator, no groupers –** In the following cases, the default separator is a copy of the specific operator that precedes the macro-call and default groupers are not used.

Left Context	Remarks
Operator	The expansion serves as the continuation of the operator-expression that begins in the left context.

This case applies to all operators (both delimiters and keywords) in the table in *Section 5.1.1, "Syntax"*.

4. **Semicolon separator, SET and TES groupers –** In the following cases, the default separator is a semicolon and default groupers are SET and TES.

Left Context	Remarks
OF	The expansion serves as the body of a case-expression or a select-expression.

This case applies to the keyword OF when it appears in a case-expression or a select-expression.

5. **Comma separator, parenthesis groupers –** In the following cases, the default separator is a comma and the default groupers are parentheses.

Left Context	Remarks
name literal attribute psect-attribute switch list-option linkage-type linkage-modifier)]	The expansion serves as a parenthesized list of actual-parameters or formal-names. This default is based on the assumption that the left context gives the address of a routine or a data segment; the usefulness of this assumption varies from one situation to another.

>
END
TES	
: (colon)	Not a useful default.
OF	The expansion serves as a repeated group of plit-items.

This case applies to the keyword OF when it appears in a plit-group.

16.3.3.5. Expansion of Keyword-Macros

The compiler uses the following algorithm for expanding a keyword macro-call:

1. **Associate actuals with formals –** Associate actual-parameters with formal-names as indicated by the keyword-assignments in the macro-call.

   If the macro-call does not include a keyword-assignment for a particular formal-name, then use the corresponding default-actual from the declaration of the macro. If the declaration does not have such a default-actual, then use the empty lexeme sequence.

2. **Complete expansion –** Complete the expansion of the macro-call as if it were a simple-macro-call (starting with step 2 of *Section 16.3.3.2, "Expansion of Simple Macros"*).

16.3.4. Discussion

The following discussion of macros begins with easy examples and continues with a section on the default punctuation of iterative macros.

16.3.4.1. Introductory Examples

Four examples of macro-declarations were given in the preceding section on macro-declarations. In the following paragraphs, each of those declarations is given again with an accompanying call and the expansion of the call.

An example of a simple-macro follows:

```
MACRO
     SM1(F1,F2,F3) =
          ((F1(F2)+F1(F3))/2) %;
...
SM1(ROUT,0,.A+.B)
```

The expansion of the call on SM1 is as follows:

```
((ROUT(0)+ROUT(.A+.B))/2)
```

In this and subsequent examples, it is assumed that the macro-call appears in a context in which it plays a valid and useful role.

An example of a conditional-macro follows:

```
MACRO
     CM1(F1,F2)[] =
          F1 = .F1 ^ -F2 ;
```

```
        CM1(%REMAINING) %;
CM1(A,0,B,6,C,2)
```

The expansion of the call on CM1 proceeds recursively, as follows. The original call yields the following:

```
A = .A ^ -0;
CM1(B,6,C,2)
```

Next, the new call is expanded, and the accumulated result is as follows:

```
A = .A ^ -0;
B = .B ^ -6;
CM1(C,2)
```

Once more the new call is expanded, as follows:

```
A = .A ^ -0;
B = .B ^ -6;
C = .C ^ -2;
CM()
```

This time, the new call has insufficient parameters, and its expansion is the null lexeme sequence, so the final result is as follows:

```
A = .A ^ -0;
B = .B ^ -6;
C = .C ^ -2;
```

The significant feature of this macro is that it can accept any number of pairs of actual-parameters, and produces an assignment for each.

An example of an iterative-macro follows:

```
MACRO
    IM1(F1)[F2] =
        F1+F2 %;
...
PLIT(IM1(2,A,B,C,D,))
```

The expansion of the call on IM1 is as follows:

```
2+A,2+B,2+C,2+D
```

Thus, in this example, the macro-call provides four plit-items for the PLIT expression.

This example illustrates two of the special features of iterative-macros. First, it shows how some parameters (just the first one in this example) can be used in each iteration of the expansion while the remaining parameters are used up (one at a time in this example) by the individual iterations. Second, it shows that the iterations are separated by a lexeme (a comma in this example) that depends on the context (a PLIT in this example).

An example of a keyword macro follows:

```
KEYWORDMACRO
    COPYVECTOR(DEST,SOURCE,N=1) =
        INCR I FROM 1 TO N DO
            DEST[.I] = .SOURCE[.I] %;
...
COPYVECTOR(N=10,DEST=V2,SOURCE=V1);
```

The expansion of the call on COPYVECTOR is as follows:

```
INCR I FROM 1 TO 10 DO
    V2[.I] = .V1[.I];
```

The main advantage of keyword macros over simple macros is that the actual-parameters need not be given in the same order as the formal-names. This is useful when the order of the formal names is hard to remember, that is, when there are many parameters or when there is no natural order. This example illustrates such a situation.

16.3.4.2. Default Punctuation

Section 16.3.3.4, "Expansion of Iterative-Macros" defines the default punctuation for iterative-macros. This section further discusses that aspect of BLISS and gives some examples.

The default punctuation of an iterative macro-call is based on an examination of the context in which the macro-call appears. The context used by the compiler is minimal (the one or two lexemes that precede the call), but it usually provides the result you want. For example:

```
MACRO
    SHIFT[A,B] = A^B %;
BIND
    PTR = PLIT(
        SHIFT(1,2,3,4,5,6),
        0+SHIFT(1,2,3,4));
```

In this example, the macro SHIFT is called twice. After expansion of these macro-calls, the BIND expression is as follows:

```
BIND
    PTR = PLIT(
        1^2,3^4,5^6,
        0+1^2+3^4);
```

The first macro-call appears after the lexeme PLIT, and should supply one or more plit-items; therefore, commas, which are the separators in a list of plit-items, are supplied as default punctuation. The second macro-call appears after the plus sign (+) lexeme, and should supply a sequence of operands; therefore, the plus sign operator, in this case, is supplied as the default punctuation.

The default punctuation is not always the punctuation you want. If you want something different, you can either avoid the use of an iterative macro or else change the context. The second macro-call in the preceding example is an example of a change of context: the zero-plus (0+) before the call changes its context without changing the value of the plit-item provided by the call.

Consider an iterative-macro-call that occurs at the beginning of a macro-actual-parameter in a larger macro-call. The iterative-macro-call is expanded prior to the containing macro-call; therefore, its context is just the left parenthesis, left bracket, or comma that precedes it in the actual-parameter list. Later, the actual-parameter replaces a formal-name in a macro-body, but that is too late to affect the expansion of the embedded iterative-macro. This aspect of macro-expansion limits the usefulness of iterative-macro-calls.

An example of default punctuation that uses default brackets arises in the processing of the following block:

```
BEGIN
MACRO
```

```
    CASEGEN(INDEX)[ ] =
        BEGIN
        MACRO
            CASELINE[ACTION] =
                [%COUNT]: ACTION %QUOTE %;
        CASE INDEX FROM 0 TO %LENGTH-2 OF
            CASELINE(%REMAINING)
        END%;
...
CASEGEN(.I, Q1, Q2, Q3);
...
END;
```

After macro expansion, this block is as follows:

```
BEGIN
...
    BEGIN
    CASE .I FROM 0 TO 4-2 OF
        SET
        [0]: Q1;
        [1]: Q2;
        [2]: Q3
        TES
    END
...
END;
```

The default brackets, SET and TES, were supplied by the compiler because the macro-call on CASELINE was expanded in the left context of OF in a case-expression.

Observe that a containing block is generated by the macro CASEGEN because it contains a nested macro-definition. The generation of a containing block is advisable for two reasons. First, the macro CASEGEN can then be called in any context, not just at the end of the declarations in a block. Second, the name of the nested macro is then confined to the scope of the generated block and is, therefore, not known at the same block level as the name CASEGEN.

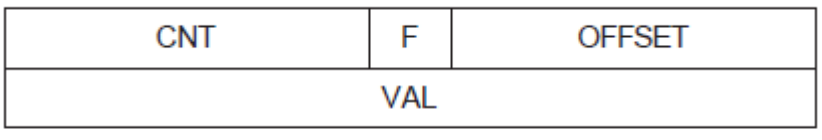
16.4. Examples of Macros

This section provides some relatively advanced examples of the use of macros. It gives some idea of the variety of tasks that macros can handle.

16.4.1. Macros for Initializing a BLOCK Structure

When a BLOCK structure is used in a program, its fields can be initialized conveniently by means of a macro. An example of this application of macros follows.

Suppose a BLISS–32 block structure that has the following layout is required:



ZK-5998-GE

Let this structure be called a QVAL block, and suppose that its fields have the following properties:

Field	Size (in Bits)	Extension
OFFSET	16	UNSIGNED
F	3	UNSIGNED
CNT	13	SIGNED
VAL	32	SIGNED

The fields are laid out in the order of increasing byte addresses, with OFFSET first, then F, and so on. Thus OFFSET occupies the first word, F occupies the low-order three bits of the second word, CNT occupies the remaining bits of that word, and VAL occupies the third and fourth 16-bit words (the entire second fullword).

The following simple-macro provides for initialization of a QVAL block:

```
MACRO
    INIT_QVAL(OFFSET,F,CNT,VAL) =
        INITIAL( WORD(OFFSET,
                 ((F) AND %O'7') OR ((CNT)^3 AND %O'177770')),
                 LONG(VAL)) %;
```

This macro packs four values, one for each field, into the correct layout for a QVAL block. Consider the following use of the macro:

```
OWN
    X: BLOCK[QVAL_SIZE] INIT_QVAL(0,3,-1,2);
```

When the macro is expanded, the declaration becomes the following:

```
OWN
    X: BLOCK[QVAL_SIZE] INITIAL( WORD(0,%O'177773'), LONG(2));
```

Observe that the values for F and CNT are packed into the second word by masking their values, shifting the CNT value three bits left, and then combining the values with an OR operator.

The use of macros described here supports the declaration and referencing of the BLOCK structures described in *Chapter 11, "Data Structures"*.

16.4.2. A Complicated Macro

Sometimes it is appropriate to use a macro for a relatively specialized and complicated purpose. An example of such an application follows:

```
MACRO
    BLOCKSETUP (A) [] =
        OWN A: BLOCK[10];
        ROUTINE %NAME (A,'_INIT'): NOVALUE =
            BEGIN
            INCR I FROM 0 TO 9 DO ! Zero the block
                %NAME (A) [.I,0,32,0] = 0;
            FILL (A, %REMAINING) ! Set fields
            END;
        %,
```

```
    FILL (A)[B] = A B %;
```

These macros declare a given name (represented by the formal-parameter A) as an OWN BLOCK composed of ten longwords. In addition, they declare a routine that, when called, initializes the block. The routine begins by setting all ten longwords to zero and then initializing any number of specified fields within the block.

Suppose that two of the fields within the block are given names as follows:

```
MACRO
    ALPHA = 0,8,8,0%,
    BETA = 5,0,16,1%;
```

It is assumed that ALPHA and BETA are the only fields that require initialization. Then an example of a call on the macro BLOCKSETUP is as follows:

```
BLOCKSETUP(QQ, [ALPHA] = 25, [BETA] = 32);
```

The expansion is as follows:

```
OWN
    QQ: BLOCK[10];
ROUTINE
    QQ_INIT: NOVALUE =
        BEGIN
        INCR I FROM 0 TO 9 DO
            QQ[.I,0,32,0] = 0;
        QQ[0,8,8,0] = 25;
        QQ[5,0,16,1] = 32;
        END;
```

Given these declarations, a call on QQ_INIT (without any actual-parameters) will zero QQ and set two of its fields.

16.4.3. Nested Macro Definition

A macro definition can be given within a macro definition, as follows:

```
MACRO
    M1(F1,F2)[] =
        OWN F1, F2;
        MACRO NM1[F3,F4] =
            LOCAL %NAME(F3,'_1'), %NAME(F4,'_1'); %QUOTE %;
        NM1(F1,F2,%REMAINING)
        %;
```

The %QUOTE lexical-function prevents the percent (%) lexeme from being lexically bound and thus from being interpreted as the termination lexeme for the macro body of M1. An example of a call on the macro M1 follows:

```
M1(A, B, C, D, E, F)
```

The result of this call is the following expansion:

```
OWN A,B;
LOCAL A_1,B_1;
LOCAL C_1,D_1;
LOCAL E_1,F_1;
```

16.4.4. Declarations Within Macros

Declarations within macros can lead to problems. For example:

```
BEGIN
MACRO
    S(A,B) =
        BEGIN
        LOCAL C;
        C = .A + .B;
        .C
        END %;
OWN C,X;
S(C,X);
S(%UNQUOTE C,X);
END
```

In the first call on S, the substitution of the actual-parameter C in the macro body causes it to be interpreted as the local variable declared in the macro body. The second call on S avoids this problem by the use of the %UNQUOTE lexical-function.

16.5. Require-Declarations

A require-declaration specifies the name of a file. When the module is compiled, the require-declaration is replaced by the contents of the file. Text that is common to a number of separate modules can be made into a single file and, in this way, included in each module (see also *Section 15.5.16, "Require-Function"*).

The most common use of a require-declaration is in connection with a file that contains structure-declarations, field-declarations, macro-declarations, and literal-declarations common to several related modules of a program.

16.5.1. Syntax

require-declaration	REQUIRE file-designator ;
file-designator	quoted-string

The syntactic name *quoted-string* is defined in *Section 4.3, "String Literals"*.

16.5.2. Restrictions

The file-designator given in a require-declaration must be a valid file name on the system on which the compiler is running.

The result of replacing the require-declaration with the specified file must be a valid module.

If the required file contains a %IF lexeme, it must also contain the matching %THEN, %ELSE (if used), and %FI of the same lexical condition.

During the expansion of a required file (declaration or function) a fatal error will occur if the end of the file is found while a macro is still being declared.

A required file (declaration or function) must not appear during the expansion of a macro.

16.5.3. Semantics

The specified file is placed at the head of the input stream. The following actions are performed:

1. Locate the file specified by the file-designator. File name default rules are given in the appropriate BLISS user manual.
2. Suspend input from the current lexeme source.
3. Adopt the specified file as the current lexeme source.
4. When the specified file is empty, resume input from the lexeme source that was suspended in step 2.

16.6. Library-Declarations

A library-declaration calls upon a file that has been *precompiled*. The effect is to introduce a set of declarations into a module without compiling them.

Before a library declaration can be compiled, a separate compilation activity must be performed. That is, a *library source file* must be created by the programmer, compiled as described in the appropriate BLISS user's guide, and saved as a *library binary file*. It is the latter file that is used when the library-declaration is compiled as part of a module.

A library-declaration (and the associated precompilation) is chosen over a require-declaration entirely for reasons of efficiency: it can reduce compilation costs. Most of the cost associated with compiling a library file is done during precompilation. Therefore a saving results if the library file is used in several modules or if it is revised less often than the modules in which it is used.

Aside from efficiency, a given library-declaration has the same effect as an analogous require-declaration.

16.6.1. Syntax

library-declaration	LIBRARY file-designator ;
file-designator	quoted-string

The syntactic name *quoted-string* is defined in *Section 4.3, "String Literals"*.

16.6.2. Restrictions

The file specified by the library-declaration must be a library binary file produced by the same compiler that is compiling the library-declaration.

The result of replacing the library-declaration with the associated library binary file must be a valid module. The compiler does not actually perform this replacement, but such a replacement is easy to imagine.

The associated library source file must not contain any use of a name that is not declared in that file.

The associated library source file must consist of a sequence of declarations. Only certain kinds of declarations can be used. These declarations, listed according to the chapters in which they are described, are as follows:

Declarations	Chapter
external-declarations	*Chapter 10, "Data Declarations"*
structure-declarations field-declarations	*Chapter 11, "Data Structures"*
external-routine-declarations	*Chapter 12, "Routines"*
linkage-declarations	*Chapter 13, "Linkages"*
external-literal-declarations literal-declarations (Specifically, LITERAL is permitted, but GLOBAL LITERAL is not) bind-data-declarations (only if data-name-value is a compile-time constant expression) bind-routine-declarations (only if routine-name-valueis a compile-time constant expression)	*Chapter 14, "Binding"*
compile-time-declarations macro-declarations keyword-macro-declarations require-declarations library-declarations	*Chapter 15, "Lexical Functions"* and *Chapter 16, "Macros"*
switches-declarations undeclare-declarations built-in-declarations	*Chapter 18, "Special Features"*

16.6.3. Semantics

The declarations encoded in the specified library binary file are incorporated into the module being compiled. The following actions are performed:

1. Locate the file specified by the file-designator. File name default rules are described in the appropriate BLISS user manual.

2. Verify that the specified file is a library binary file and that the compiler that generated the file is compatible with the compiler that is compiling the library-declaration.

3. Add the precompiled tables that make up the specified file to the tables already formed by the compiler.

The result is to establish a set of declarations with a minimum of compiler activity.

Switches-declarations in the library source file affect the precompilation of the file but have no effect on the module that uses the file in a library-declaration.

Lexical-expressions are expanded at the time a library source file is compiled to produce the library binary file, not when the library binary file is incorporated into another module.

The undeclare-declaration can be used at the end of a library source file to prevent declarations from being output to the library binary file. In this way, the effect of a declaration can be confined to the compilation of the library file itself. This approach is essential when the same name is declared in several library files that are used together in the same module.

A library source file can include both a require-declaration and a library-declaration.

Library declarations are permitted in a library precompilation to allow data-structuring packages (such as XPORT) to be used both in library construction and within any individual modules that refer to the library.

All symbols defined by the nested library will be implicitly undeclared at the end of precompilation; this prevents the generation of error messages due to names being declared in two libraries. However, if it is necessary to retain the symbols from the declared library, the library can be referenced by a require-declaration using the source file as file-designator.

As an example, assume library COMLIB is being built to contain a common set of data structures for a project; moreover, the structures use XPORT $FIELD macros, while the project uses the XPORT I/O package. Thus, COMLIB.REQ will contain lines such as the following:

```
LIBRARY 'SYS$LIBRARY:XPORT';

...

$FIELD
    LINKED _LIST=
        SET
        NEXT=   [$ADDRESS]
        LAST=   [$ADDRESS]'
        VALU=   [$INTEGER]
        TES;

...
```

When COMLIB.REQ is being precompiled, the $FIELD, $ADDRESS, and $INTEGER definitions are defined by the XPORT library; however, at the end of the precompilation process the definitions are deleted.

When a module that uses XPORT I/O is compiled it can contain the following lines:

```
LIBRARY 'SYS$LIBRARY:XPORT';
LIBRARY 'LIB$:COMLIB';

...
```

Note that if the COMLIB library contained a macro declaration such as the following:

```
MACRO DOLLAR_FIELD = $FIELD %;
```

the macro would not be expanded at declaration time and $FIELD would be unbound. Thus, if a source module (that did not have a library XPORT declaration) referenced the DOLLAR_FIELD macro, $FIELD would be treated as an undefined name.

Another example of a library-declaration within a library compilation follows. This example emphasizes the sometimes unexpected behavior that can occur during the compilation of nested libraries.

For the example, assume that two files are separately compiled as follows:

```
$ BLISS/LIBRARY INNER
.
.
.
$ BLISS/LIBRARY OUTER
```

The first compilation produces INNER.L32 as follows:

```
;   0001  0     FIELD
;   0002  0         CAB_FIELDS =
;   0003  0             SET
;   0004  0             CAB$W_BLN        = [1,2,3]
;   0005  0             TES;
```

The second compilation produces OUTER.L32 as follows:

```
;   0001  0     LIBRARY 'INNER';
;   0002  0
;   0003  0     EXTERNAL ZOT  : BLOCK[100] FIELD(CAB_FIELDS);
; WARN 201
; Illegal occurrence of bound name CAB_FIELDS in library source
; module
```

The error message occurs because symbols from the INNER library are not included in the OUTER library. The symbol ZOT, declared in the OUTER library, refers to the symbol CAB_FIELDS, declared in the INNER library; if, in a subsequent compilation, the OUTER library is used without the INNER library the declaration of CAB_FIELDS will not be available.

Chapter 17. Condition Handling

Condition handling is the response to an unusual event that is signaled during execution of a program. The unusual event is often the detection of an error, but need not be; it could, for example, be part of a scheme to measure the performance of a program. This chapter describes the features of BLISS that support condition handling.

Condition handling involves the BLISS language together with the target hardware and software system. For additional system details, see the respective hardware and operating system reference manuals, as well as the respective BLISS user manuals.

17.1. Introduction to Condition Handling

Condition handling begins when an event or situation is signaled by a call on one of the executable-functions SIGNAL or SIGNAL_STOP. The signal is directed to a part of the system called the Condition Handling Facility (CHF). The CHF retains control until the unusual event has been dealt with; but the CHF can, and usually does, call user routines for assistance. Then, depending on the outcome, program execution continues or is terminated.

17.1.1. Routines

Condition handling involves the interaction of three kinds of routines. First is a *signaler routine*, which contains code that generates the signal, either explicitly or implicitily. Second are *handler routines*, which are called upon by the CHF to provide the desired response to a signal. Third are *establisher routines* that contain a special declaration, the *enable-declaration*, that associates a handler routine with the establisher routine.

These three kinds of routines are not new; they are routines that are used in a new way, to play special roles in condition handling. A single routine can play two or three of these roles at the same time; in fact, a routine can even establish itself as its own condition handler.

Furthermore, a single routine can be used in many places; for example, a single routine can be established as the handler routine by many establisher routines.

17.1.2. Signals

A signal can be generated in three ways. First, a signal can be explicitly generated by a call on the executable-function SIGNAL or SIGNAL_STOP. Second, a signal can be implicitly generated by the hardware or the software system as a result of a condition detected during program execution. Third, a special kind of signal, the unwind signal, can be indirectly generated by a handler routine by means of a call on the executable-function SETUNWIND.

When a signal is generated, a data segment termed the *signal vector* is used to describe the condition. This vector contains a *condition value*, which is an encoding of the primary description of the condition that caused the signal. The encoding of the condition value is defined by software conventions and is the same for all conditions. The remaining elements of the signal vector provide suplementary information about the condition; this information can vary from one condition to another.

17.1.3. Processing

When condition handling is initiated, the CHF searches the stack of routine calls for the most recently established handler. The handler is called by the CHF with three parameters giving, respectively, values

from the signaler (one of which is a condition value), values from the CHF itself, and values from the establisher of the handler. The handler uses this information to determine what action to take in response to the condition.

The handler indicates to the CHF how condition handling for the signal should proceed after the handler returns to the CHF. In the simplest case, the handler requests the CHF to return to the signaler. This completes condition handling for that signal.

The handler can also request resignaling. In this case, CHF searches for the next handler in the stack of routine calls and calls it. The search for and calling of successive handlers continues as long as each handler in turn requests resignaling.

Finally, the handler can request unwinding. Unwinding causes the execution of various routines to be terminated by removing each routine's stack frame from the stack of routine calls as though the routine had returned normally.

During unwinding, the handler of any routine that is being terminated is called (a second time) to give each handler an opportunity to perform any actions necessary on behalf of the establisher in order for the establisher to complete properly. Examples of such actions are closing files opened by the establisher, releasing dynamically allocated storage, adjusting counters and flags, and so on. Normal execution resumes after the call to the establisher of the handler that requested unwinding. This completes condition handling for that signal.

The description of condition handling is given in five parts. The first three parts present the BLISS language features relevant to the three kinds of routines involved in condition handling. First, enable-declarations, used in establisher routines, are described. Second, signals and the means by which a signaler routine initiates condition handling are described. Third, handler routines, their parameters and the means by which a handler directs CHF processing are described. The fourth part describes the flow of control during condition handling among the three kinds of routines. The fifth part gives examples of the application of condition handling.

17.2. Enable-Declarations

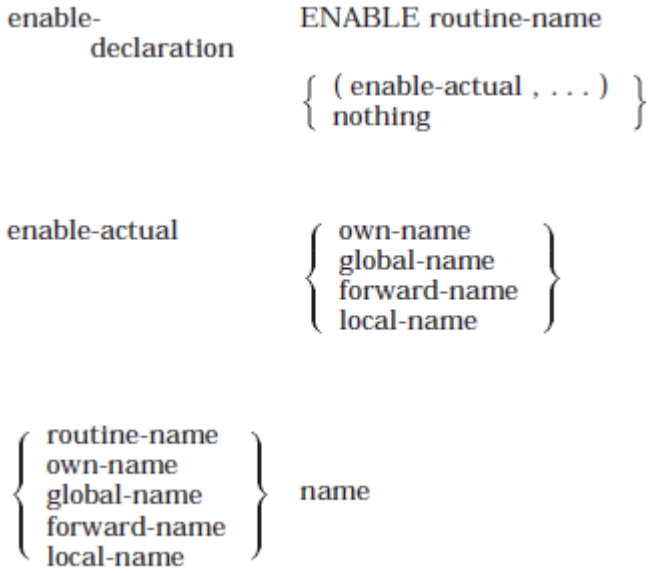
An enable-declaration is the means by which one routine, an establisher, identifies another routine as a handler routine. The association is established at the beginning of the establisher's execution and lasts throughout the execution of that routine and any routines that it calls. The association is automatically broken when the establisher routine returns.

In addition to specifying the handler routine, the establisher can also specify parameters that will be passed to the handler if the handler is actually called. For example:

```
ROUTINE X(Y,Z) =
    BEGIN
    EXTERNAL ROUTINE
        XH;
    LOCAL
        L: VOLATILE;
    ENABLE
        XH(L);
    . . .
    END;
```

Routine X establishes the routine XH as its handler and specifies the address of a local data segment, L, to be passed to the handler when the handler is called.

17.2.1. Syntax



17.2.2. Restrictions

An enable-declaration must appear only in the outermost block of a routine-definition.

Only one enable-declaration can appear in the outermost block of a routine-definition. This does not prohibit a nested routine, as well as the outer routine, from containing an enable-declaration.

In BLISS–16 and BLISS–32, a routine that contains an enable-declaration must be declared with a linkage-attribute that is itself declared with a linkage-type as follows: the JSR linkage-type in BLISS–16, or the CALL linkage-type in BLISS–32; observe that the predeclared default linkage satisfies this restriction in each case. Further, no external registers or output-registers are permitted.

The routine-name given in an enable-declaration must be the name of a routine declared in a routine- or bind-routine-declaration.

In BLISS–16 and BLISS–32, the linkage-attribute of the handler routine-name given in the enable-declaration must be the predefined linkage-attribute BLISS.

Each data segment name that appears as an enable-actual parameter in an enable-declaration must have the volatile-attribute specified in its declaration.

If the handler routine can potentially modify any data segment other than an enable-actual data segment (for example, a data segment whose address is given by the contents of an enable-actual parameter), that data segment must be declared with the volatile-attribute.

17.2.3. Semantics

The enable-declaration establishes a given routine as the routine to handle any software- or hardware-detected conditions that are signaled during the execution of the routine containing the enable-declaration. The execution of the establisher includes the execution of any routines that it calls, directly

or indirectly. However, it may or may not include the execution of any handlers, as described in *Section 17.5, "Condition-Handling Flow of Control"*.

The enable-actual parameters given in the declaration are the names of data segments whose address values are passed to the handler when and if it is called.

An enable-actual parameter can be the name of a local data segment (declared LOCAL or STACKLOCAL) and if so, that data segment is implicitly initialized to all zero bits before the handler routine is established.

The enable-declaration does not, of itself, call the given handler routine.

17.3. Signaling

Signaling initiates condition handling and thereby indicates that a particular event or condition has occurred. You can explicitly generate a signal by calling one of the executable-functions SIGNAL or SIGNAL_STOP. Such signals can be implicitly generated by hardware-detected error conditions (such as an access violation or arithmetic overflow) and can be indirectly generated by a handler routine request for unwinding.

All signals identify a condition by means of a vector that contains a condition value. The vector can also contain additional values that provide auxiliary information about the condition.

17.3.1. Condition Values

A *condition value* is a single fullword value that encodes the identity and severity of the condition. The severity field is encoded in the low-order three bits and the identity field in the remaining high-order bits. In BLISS–16, the identity field consists of all 13 of the high-order bits of the 16-bit word. In BLISS–32 the identity field consists of the next 25 bits (above the severity field), and in BLISS–36 consists of the next 29 bits, leaving the high-order four bits for other purposes in both dialects.

When accessing a condition value to determine which condition is being reported, it is necessary to examine only the identity field, excluding the remainder. The same condition identity value may be signaled with different severity values at different times.

A more detailed description of condition value representation is given in *Section 17.6.1, "Accessing and Defining Condition Values"*, along with example declarations for conveniently creating and accessing condition values.

17.3.2. Explicit Signals

BLISS programs can explicitly generate a signal by calling one of the executable-functions SIGNAL or SIGNAL_STOP. These functions are defined as follows:

SIGNAL(***condition-value*** **)**
SIGNAL(***condition-value*** **,** ***parameter*** **, . . .)**

> Initiates condition handling for the condition indicated by the given condition-value. If parameters are given in addition to the condition-value, these values are included in the signal vector (see *Section 17.4.2.1, "The Signal Parameter"*) passed to each handler that is called.
>
> The function returns if and only if a handler for the condition requests continuation. In BLISS–32, the VMS system establishes a default catchall handler for all signals; see *Section 17.6.4.2, "The VMS Operating System"*.

The function returns a value if and only if a handler assigns a returned-value to the mechanism vector (see *Section 17.4.2.2, "The Mechanism Parameter"*); otherwise, the value is undefined.

SIGNAL_STOP(***condition-value*** **)**
SIGNAL_STOP(***condition-value*** **,** ***parameter*** **, . . .)**

Initiates condition handling for the condition indicated by the given condition-value. A condition-value with the severity field replaced by the code for severe error (STS$K_SEVERE, see *Section 17.6.1, "Accessing and Defining Condition Values"*) is included in the signal vector passed to each handler that is called. If parameters are given in addition to the condition-value, these values are also included in the signal vector passed to each handler. The function does not return a value.

SIGNAL and SIGNAL_STOP are identical in their actions, with two exceptions. First, if SIGNAL is called, control may eventually return to the caller depending on the actions of the handler, while if SIGNAL_STOP is called, control will not return to the caller. Second, the condition-value of a SIGNAL_STOP call is changed to indicate a severe error while the condition-value of a SIGNAL call is used without modification.

Information can be returned from a handler to a signaler if the signaler includes a parameter in the call to SIGNAL that gives the address of a data segment where the information should be assigned by the handler.

17.3.3. Implicit Signals

Signals may be generated by the system in response to a hardware detected condition or an operating system detected condition. For hardware conditions, the system uses the information available from the hardware and simulates a call to SIGNAL as though SIGNAL were called at the instruction that caused the error (either before or after the instruction, depending on the target system and the type of hardware condition). Thereafter, processing is the same as for explicitly generated signals.

17.3.4. Unwind Signals

The handler of a condition may cause the routine that generated a signal to be terminated. In fact, many routines may be terminated in this abnormal way, called *unwinding*. During unwinding, the handler of each routine that is being terminated is called with a condition value indicating that the establisher routine is being terminated. This particular condition is termed the *unwind signal* and some special rules apply.

Unwind signals are further discussed in the next section.

17.4. Condition-Handling Routines

A condition-handling routine is a routine that is declared by some other routine to be a handler. The purpose of a condition-handling routine is to accept and deal appropriately with some set of signaled conditions that may occur during the execution of the establisher. In nearly all respects, a handler routine is like any other routine: it can call other routines, call the operating system for service, and so on. It can establish a handler for itself and in some cases that handler might even be itself.

A handler is special in that it is called in response to conditions that are signaled by other routines. It is unlikely that a routine written for use as a handler would ever be called directly. Because handlers are called by system software, and not directly by user-written calls, they must conform to system-defined restrictions and conventions.

A handler is called by the CHF with three actual parameters. The first parameter is the address of a vector, termed the signal vector, that contains the parameter values specified in the call to SIGNAL or SIGNAL_STOP that generated the trace signal. In BLISS–32, additional values are supplied as well. The second parameter is the address of a second vector, termed the mechanism vector, that contains values provided by the CHF software. The third parameter is the address of a third vector, termed the enable vector, that contains the enable-actual parameter values specified in the enable-declaration of the routine that established the handler. Thus, a handler has available information from both the routine generating the signal and the routine that established the handler, as well as certain system information, to determine how to deal with the condition.

A handler is called as a result of every signal that occurs during the execution of its establisher and that is not dealt with by another handler. The first responsibility of every handler is to examine the condition value of each signal to determine whether the signal is to be dealt with at all. It is quite unusual for a specific handler to be relevant to every possible signal that can occur.

If a signal is not the unwind signal, a handler must request the CHF to further process the signal in one of the following ways:

- Continue the routine that generated the signal.
- Resignal the same signal, or possibly a modified signal, to some other handler.
- Unwind.

The next three sections discuss condition handling routines in detail. The first specifies the restrictions that must be met by every handler routine. The second describes the parameters to a handler routine. The third specifies how a handler routine requests each of the three options.

17.4.1. Restrictions

1. In BLISS–16 and BLISS–32, a condition-handling routine must be declared with the predeclared linkage-attribute BLISS (see *Section 13.5, "Common Predeclared Linkage-Names"*). Observe that this will be the default unless another default is established by a LINKAGE switch-item (see *Section 18.2, "Switches-Declarations"*) or module-switch (see *Section 19.2, "Module-Switches"*).

   A condition-handling routine must be declared with three formal parameters.

   A condition-handling routine must not have the NOVALUE attribute unless it always requests unwinding for every signal.

   A condition handling routine must fetch from or assign to data segments that satisfy one of the following requirements:

   - A data segment whose scope is limited to the body of the condition handling routine itself
   - An element of one of the vectors whose addresses are passed to the handler as parameters
   - Any data segment that is declared with the volatile-attribute

17.4.2. Parameters

A condition handling routine is called with three parameters. Each parameter is the address of a counted vector containing the relevant information. A *counted vector* is a vector of fullwords in which the first

element (with index value zero) contains the number of additional elements in the vector. The first element is always present and contains the value zero if there are no additional elements in the vector.

The following BLISS code fragment shows a template for the declaration of a handler routine. This template is used in the remainder of this section in the discussion of each parameter of a handler routine. The template is as follows:

```
ROUTINE HANDLER(SIG, MECH, ENBL) =
    BEGIN
    MAP
        SIG: REF VECTOR,      ! Signal vector
        MECH: REF VECTOR,     ! Mechanism vector
        ENBL: REF VECTOR;     ! Enable vector
    BIND
        COND = SIG[1]: CONDITION_VALUE,
        RETURN_VALUE = MECH
            [
            %BLISS16(1)
            %BLISS36(1)
            %BLISS32(3)
            ];
...
END;
```

In this template, the map-declaration (see *Section 10.10, "Map-Declarations"*) associates the REF VECTOR structure-attribute (see *Section 11.10.1, "VECTOR Structures"*) with each of the routine formal names for referencing of each vector whose address is passed to the handler. The bind-declaration (see *Section 14.3, "Bind-Data-Declarations"*) defines names for two of the most commonly accessed elements of the passed vectors. CONDITION_VALUE is the name of a macro whose expansion gives the attributes appropriate for accessing a condition value. Its definition is presented in *Section 17.6.1, "Accessing and Defining Condition Values"*. The predeclared macros %BLISS16, %BLISS32, and %BLISS36 are described in *Section 16.2.4, "Predeclared Macros"*.

17.4.2.1. The Signal Parameter

The first parameter, SIG, contains the address of a *signal vector*, which is a counted vector that contains the values of the actual parameters of the call to SIGNAL or SIGNAL_STOP. In BLISS–32, the CHF adds two values following those given in the SIGNAL or SIGNAL_STOP call: the hardware program counter (PC) and the program status longword (PSL) of the next instruction to execute in case the handler requests continuation of the signaler. In the context of the above template, .SIG[n] is the nth actual parameter value, where, in particular, .SIG[1] is the condition value, .SIG[0] is the number of actual parameters, and COND is the address of the condition value.

For explicit signals, the actual parameter values are given by the parameters of the call to SIGNAL or SIGNAL_STOP.

For implicit signals and the unwind signal, the actual parameter values are defined by the system. These values and their encodings are not described in this manual.

17.4.2.2. The Mechanism Parameter

The second parameter, MECH, contains the address of a *mechanism vector*, which is a counted vector that contains the values of parameters provided by the CHF. These values provide specialized software status information about the signal being processed. Of the several values that may be present, only one is described in this manual.

The element of the mechanism vector with address MECH[1] in BLISS–16 and BLISS–36, or MECH[3] in BLISS–32 in the preceding template, is a data segment to which a handler routine can assign a value to be used as a returned-value. A handler can assign a value to this location in two situations.

When a handler requests continuation of the signaler routine, the CHF uses the contents of this location as the return value of the SIGNAL call. By assigning to this location, the handler can determine the return value. If the handler does not assign to this location, the returned value is undefined.

After unwind processing, the CHF uses the contents of the return value location in the mechanism vector as the return value of the last routine to be terminated. By assigning to this location, the handler can determine the establisher's return value. By this means, the establisher routine returns a meaningful value to its caller even though it is terminated by the CHF. A handler for any establisher that returns a value (that is, does not have the NOVALUE attribute) must assign an appropriate return value to the return value location in the mechanism vector during unwinding.

17.4.2.3. The Enable Parameter

The third parameter, ENBL, contains the address of an *enable vector*, which is a counted vector that contains the values of the enable-actual parameters of the ENABLE declaration of the establisher routine. In the context of the earlier template, the expression .ENBL[n] is the nth enable-actual parameter value, and .ENBL[0] is the number of enable-actual parameters.

The enable-declaration requires that each enable-actual parameter must be the address of a data segment. Consequently, within the handler routine it may frequently be convenient to bind (*Section 14.3, "Bind-Data-Declarations"*) names to these address values, as in the following:

```
BIND
    PARAM1 = .ENBL[1],
    XYZZY = .ENBL[2];
```

Enable-actual parameters can be the names of local data segments declared in the establisher routine. If a recursive routine establishes a handler, the same handler will be used for all active calls of the recursive routine. If the handler is called and resignals the condition, the same handler is repeatedly called for each active call of the establisher routine. In each case, the address of a local data segment name passed to the handler is the appropriate address in the respective active call of the establisher.

17.4.3. Handler Options

For every condition other than the unwind signal, a handler must request one of three subsequent actions for the CHF to perform after the handler returns.

1. The handler can deal appropriately with the condition and then cause the routine that initiated the signal to *continue*.

   Continuing the routine that initiated the signal completes processing of the condition.

2. The handler can *resignal* using the same condition value, or possibly a modified one.

   Resignaling with the same condition value is the normal response for a condition that the handler does not deal with. Resignaling causes the CHF to resume searching for a handler that will deal with the condition.

3. The handler can deal appropriately with the condition and then terminate the execution of the routine that generated the signal as well as the other routines called by the establisher by *unwinding*.

Unwinding causes a special unwind signal to be generated. The handlers of all routines that are being terminated will be called with this condition. Unwinding also completes processing of the condition.

These options are not available when a handler is called for the unwind signal.

The means of requesting these actions are presented in the following sections.

17.4.3.1. Continuation

A handler requests continuation of the routine that generated the signal by returning a true value (low bit set to 1) to the CHF. The handler must not also call SETUNWIND, as described in *Section 17.4.3.3, "Unwinding"*.

After the handler returns to the CHF, the CHF returns from the call to SIGNAL in the routine that generated the signal.

A handler must not request continuation for a signal that was generated by calling SIGNAL_STOP. That is, a handler must not request continuation if the severity field of the condition-value indicates severe error.

17.4.3.2. Resignaling

A handler requests resignaling by returning a false value (low bit set to 0) to the CHF. The handler must not also call SETUNWIND, as described in *Section 17.4.3.3, "Unwinding"*.

After the handler returns to the CHF, the CHF searches for another handler routine to call, as described in *Section 17.5, "Condition-Handling Flow of Control"*.

When resignaling is requested, the same signal vector is passed to subsequent condition handlers that are called. Thus, the severity and the condition identification can be changed when the handler assigns new values to the condition value element of the signal vector. If condition handling is initiated by a SIGNAL_STOP call, however, the severity field is set to severe error by the CHF each time a handler is called. Consequently, the severity field cannot be changed by a handler in this case.

Changing the condition value and resignaling is different from generating a new signal by calling SIGNAL or SIGNAL_STOP in the handler. In the latter case, processing of the first signal is suspended until processing of the second signal is completed; then processing of the first signal resumes.

17.4.3.3. Unwinding

A handler requests unwinding by calling the executable-function SETUNWIND. The function is defined as follows:

```
SETUNWIND( )
SETUNWIND( parameter )                ! 32 Only
SETUNWIND( parameter , parameter )    ! 32 Only
```

Requests the CHF to initiate unwind processing after the currently executing handler returns to the CHF. In BLISS–32, the two optional parameters can be used to specify the routine level at which the unwind will stop and the address where normal execution is to resume. These parameters are not described in this manual. The function does not return a value in BLISS–16 or BLISS–36, and returns a VMS-defined status value in BLISS–32.

When a handler requests unwinding the returned-value of the handler is ignored.

The handler specifies the value to be used as the returned-value of the establisher by assigning the appropriate value in the mechanism parameter vector (see *Section 17.4.2.2, "The Mechanism Parameter"*) when the handler is called for the unwind signal.

In the default case, that is, when no parameters are given in the call to SETUNWIND, all routines between and including the routine that generated the signal and the establisher of the handler are terminated. Execution resumes after the call to the establisher as though the establisher had returned in the normal way.

Unwinding does not start immediately when SETUNWIND is called. The call simply advises the CHF that unwinding is requested. When the handler eventually returns to the CHF, unwinding begins.

During unwind processing, the handler, if any, of each routine being terminated is called with a condition value indicating an unwind is in progress. In the default case, where the establisher is one of the routines being terminated, the handler requesting the unwind will itself be called a second time to process the unwind signal.

A condition handling routine can call other routines as part of its processing and the request for unwinding can be made from any such routine. The call to SETUNWIND need not be made in the topmost routine directly called by the CHF.

It is invalid to request unwinding in any of the following cases:

- Condition handling is not in progress.
- An unwind request has already been made.
- Unwind signal processing is in progress.

17.5. Condition-Handling Flow of Control

Condition-handling flow of control refers to the order in which condition handling routines are called during condition handling. The order is defined in terms of the stack of routine calls that are active at the time a signal is generated in combination with subsequent handler requests.

17.5.1. Definition

The definition of condition-handling flow of control has two parts. The first part defines the flow of control for a signal that is generated when condition handling is not in progress. The second defines the modified flow of control that results for a signal that is generated while condition handling for a previous signal is still in progress.

17.5.1.1. Normal Flow of Control

The generation of a signal begins a sequence of events that is carried out under the control of the CHF.

First, the CHF creates the signal vector and mechanism vector for use in calling a handler. If the signal is generated by a SIGNAL_STOP call, the severity field of the condition value in the signal vector is assigned the code for severe error.

Next, the stack of routine calls is searched, beginning with the routine that generated the signal. If that routine did not establish a handler, then the routine that called it is considered, and so on, until the most recently called routine is found that did establish a handler. This handler is called with three parameters, as described in *Section 17.4.2, "Parameters"*.

Following the return from the handler, processing depends upon which option is requested by the handler.

If continuation is requested, then the CHF returns to the signaler and condition handling for that signal is completed.

If resignaling is requested, then the CHF continues searching the stack prior to the establisher of the handler just called. If another handler is found, then it is called in the same way as the previous handler. This process of searching for and calling successive handlers continues as long as each handler requests resignaling. If every handler indicates resignaling, that is, no handler is found that causes completion of the signal, then system defined error processing takes place.

In BLISS–16, if no handler is found the program exits. In BLISS–36, if no handler is found a message is displayed on the user's terminal and the program exits.

In BLISS–32, the VMS system establishes a catchall handler to provide default handling for all signals. Consequently, this handler will be called if no user handler is found or if every user handler requests resignaling. The action of this handler is described in *Section 17.6.4.2, "The VMS Operating System"*.

If unwinding is requested, then the handler just called and its establisher are remembered and a new search is started. This search starts over at the signaler routine just as in the first search. This time, however, each routine is terminated by removing its stack frame from the stack of routine calls. If the routine has a handler, the routine is terminated after the handler is called with a condition value that indicates that unwinding is in progress. The handler does not have the three options that are available during the first search: SETUNWIND must not be called and the value of the handler is ignored. This second search completes after the handler that initiated unwinding is called the second time. When that handler returns, the establisher is terminated and normal execution resumes immediately following the call to the establisher.

17.5.1.2. Modified Flow of Control for Nested Signals

A *nested signal* is a signal that is generated while condition handling for a previous signal is in progress. A nested signal occurs, for example, if a handler routine calls SIGNAL. When a nested signal is generated, condition handling for the previous signal is suspended until condition handling for the nested signal is completed. Then processing resumes for the previous signal.

Processing of a nested signal is the same as for a nonnested signal with one exception: the search for handlers is modified to exclude any handlers that have been called for the previous signal. The handler that is active when the nested signal is generated is excluded by this rule. However, this handler can itself have a handler and if so, this (second) handler is included in the modified search.

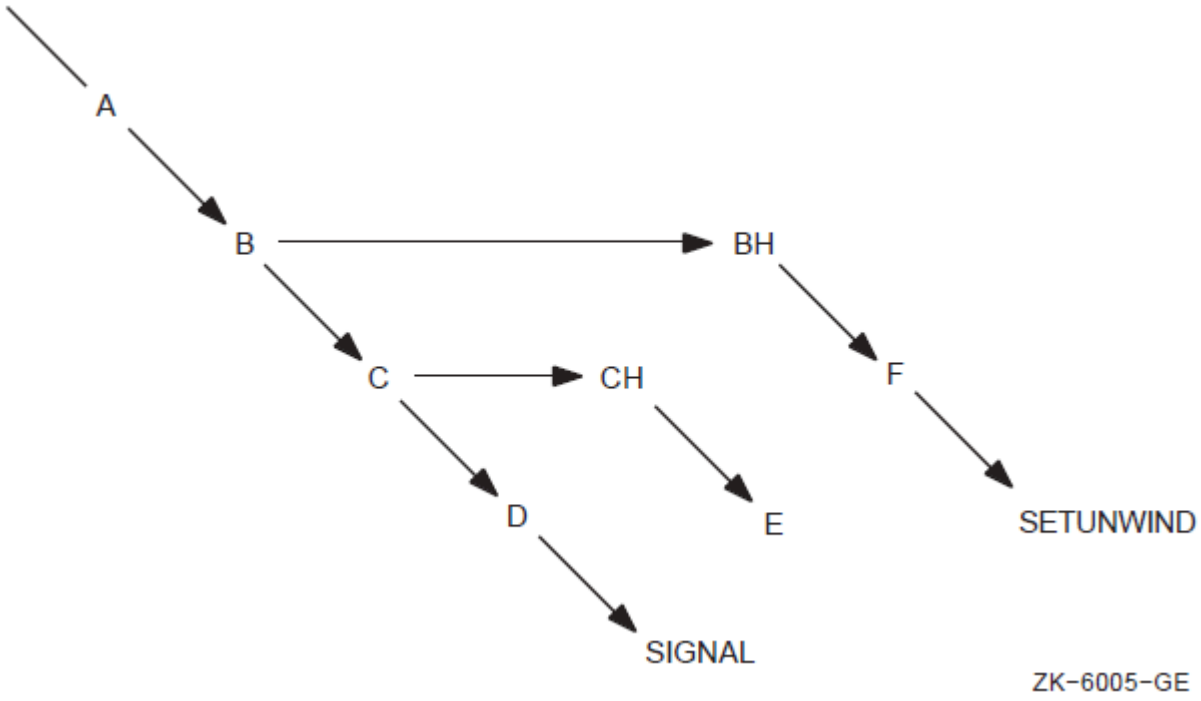
If the handler of a previous signal is terminated (so that it cannot request CHF processing) because of an unwind request for a nested signal, then all of the routines considered during condition handling of the previous signal are also terminated. The handlers of the combined set of routines being terminated are all called with the unwind signal in the inverse order to which they were established. More than one previous signal can be affected in this way. Completion of unwinding completes condition handling for all of the affected signals.

17.5.2. Discussion

Several aspects of condition handling flow of control are discussed. First, examples of the detailed sequence of events are illustrated. Second, recursive handlers are considered. Finally, interactions between condition handling and routine linkages are discussed.

17.5.2.1. Examples of Flow of Control

Example sequences of flow of control during signal processing are illustrated using the following diagram:



In this diagram, a diagonal line indicates that the upper routine calls the lower routine, for example, A calls B, B calls C, and so on. A horizontal line indicates that the left routine establishes the right routine as a handler; for example, routine C establishes routine CH as its handler.

The example begins by assuming that routine A is executing, that is, A has been called by some other routine not shown in the diagram.

Routine A does not establish a handler. At some point in its execution A calls routine B. B establishes routine BH as a handler; BH is not called when it is established. B calls routine C. Routine C establishes handler CH and then calls D. D does not establish a handler but does generate a signal.

At this point the stack of routine calls consists of A, B, C, and D, with D being the most recently called (the call to SIGNAL does not count). Routines B and C have established handlers, but A and D do not.

The CHF searches for a handler. First routine D is considered, but no handler is established. Next, routine C is considered. A handler is established and, thus, CH is called. CH calls another routine E which returns to CH which returns to the CHF. What happens next depends on the option requested by CH.

First, suppose that CH requests continuation. In this case, the CHF returns to D and D continues. The complete sequence of events is summarized as follows:

A calls B
B establishes handler BH
B calls C
C establishes handler CH

C calls D
D calls SIGNAL
CHF calls CH
CH calls E
E returns to CH
CH returns to CHF requesting continuation
CHF returns to D
D continues

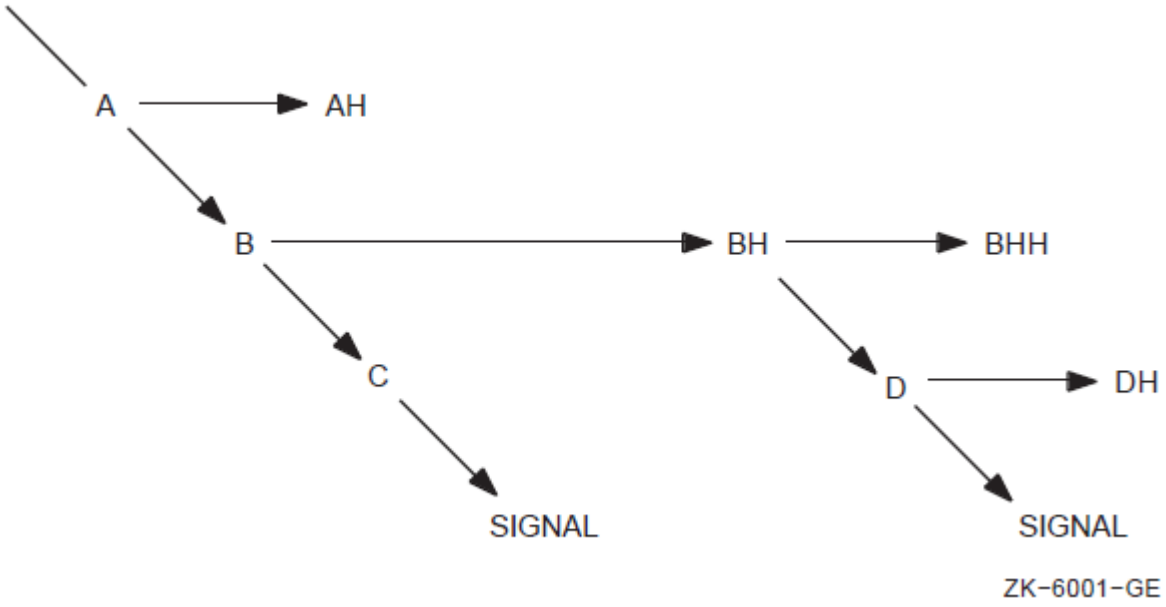
Next, suppose that CH requests resignaling (instead of continuation). In this case, the CHF continues searching for a handler by considering routine B. B has a handler, and, thus, BH is called. BH calls F and F calls SETUNWIND. The CHF records the fact that an unwind is requested and returns to F. F returns to BH and BH returns to CHF. The value of BH is not used by CHF because unwinding has been requested. At this point, the second search starts. D does not have a handler and is terminated. CHF call CH which returns to CHF. C is terminated. CHF calls BH which returns to CHF. This completes the second search. B is terminated. Finally, CHF "returns" to A using the returned-value obtained from the mechanism vector as though B had returned in normal fashion and A continues.

The sequence of events is summarized as follows (the first nine events are the same as the preceding summary):

A calls B
B establishes handler BH
B calls C
C establishes handler CH
C calls D
D calls SIGNAL
CHF calls CH
CH calls E
E returns to CH
CH returns to CHF requesting resignaling
CHF calls BH
BH calls F
F calls SETUNWIND
CHF records the unwind request
CHF returns to F
F returns to BH
BH returns to CHF
D is terminated
CHF calls CH with the unwind signal
CH returns to CHF
C is terminated
CHF calls BH with the unwind signal
BH returns to CHF
B is terminated
CHF "returns" to A as though B had returned
A continues

Observe in this example that handler BH must assign the return value of B for the call from A when BH is called for the unwind signal. If BH assigned the return value the first time it was called, there is the possibility that some other handler, such as CH in this example, will assign a return value when it is called with the unwind signal. Thus, the returned value intended by BH would be lost.

For an example of nested signal processing, the following diagram is used:



The initial sequence of events is apparent from the previous examples and is summarized by the following:

A establishes handler AH
A calls B
B establishes handler BH
B calls C
C calls SIGNAL
CHF calls BH
BH establishes handler BHH
BH calls D
D establishes handler DH

At this point D generates a nested signal. The modified search in this case considers, as potential establishers, only routines D, BH, A, and so on. Routines C and B are excluded from consideration. Assume that DH and BHH request resignaling and AH requests continuation. Events proceed as follows:

D calls SIGNAL
CHF calls DH
DH returns to CHF requesting resignaling
CHF calls BHH
BHH returns to CHF requesting resignaling
CHF calls AH
AH returns to CHF requesting continuation
CHF returns to D

At this point, processing of the nested signal is complete and processing of the first signal resumes. The subsequent sequence of events is not described here.

As a final possibility, assume for the nested signal just illustrated that DH and BHH request resignaling (as before) and AH requests unwinding (instead of continuation). In this case, control will not return to D or BH because they will be terminated. Consequently, BH cannot request an option for the first signal. Processing of the first signal must, consequently, be terminated as well. In effect, the unwind requested by AH for the nested signal also applies to the previous signal. This can apply to yet a third signal if

the previous signal was itself a nested signal, and so on. The second search of the stack considers all of the routines that are being terminated including those so far considered by the first signal. In this example, the order of consideration is D, BH, C, B, and A. Events proceed as follows (starting when AH is called):

AH calls SETUNWIND
CHF records unwind request
CHF returns to AH
AH returns to CHF
CHF calls DH with the unwind signal
DH returns to CHF
D is terminated
CHF calls BHH with the unwind signal
BHH returns to CHF
BH is terminated
C is terminated
CHF calls BH with the unwind signal
BH returns to CHF
B is terminated
CHF calls AH with the unwind signal
AH returns to CHF
A is terminated
CHF "returns" to A's caller (not shown)
A's caller continues

17.5.2.2. Recursive Handlers

A *recursive handler* routine is a handler routine that establishes itself as a handler or that calls (directly or indirectly) another routine that establishes it as a handler. Consequently, it is possible during the execution of such a handler that it will be recursively called to handle a nested signal.

Programming a recursive handler can be more difficult than programming a nonrecursive handler, just as programming any recursive routine can be more difficult than a nonrecursive routine. You must consider the sequence of events that may result from the combination of the two (or more) calls of the same routine.

Observe that each call of the handler will be caused by a different signal.

17.5.2.3. Condition Handling and Linkage Interactions

The flow of control during the processing of a signal causes various routines to be called in an order that may not be apparent when examining a program. The CHF software depends on calling sequence conventions to ensure proper accounting for the machine registers and other machine status values during this process.

The linkage-declaration (see *Section 13.3, "BLISS–32 Linkage-Declarations"*) provides the ability to choose many calling sequence variations other than the predefined linkages BLISS and FORTRAN. When you use such nonstandard linkages, there are various complex rules and restrictions that must be followed.

In BLISS–32, note that a routine whose linkage-attribute is defined with JSB linkage-type must not contain an enable-declaration and must not be declared as a handler. Such routines cannot directly interact with the CHF software, except to call the functions SIGNAL, SIGNAL_STOP, or SETUNWIND.

17.6. Examples

The following sections give examples of applying various aspects of condition handling. Because condition handling involves the interaction of several routines, complete examples are necessarily quite lengthy. The examples given below leave out many details in order to be as brief as possible.

The first section presents declarations that are suitable for accessing and creating condition values. The following sections illustrate applications of condition handling.

17.6.1. Accessing and Defining Condition Values

Condition values have similar but not identical encodings in BLISS–16 and BLISS–32. The following two sections give the encodings used and declarations for conveniently accessing and defining condition values, in BLISS–16 and BLISS–32 respectively.

17.6.1.1. Condition Values in BLISS–16

In BLISS–16, a condition value is a single fullword value that is encoded with two primary fields: a severity field in the low-order 3 bits, and an identity field in the high-order 13 bits.

The identity field is itself divided into two fields: the condition identification field and the customer definition flag.

The twelve low-order bits of the identity field (bits 3 through 14 of the condition value) are the condition identification field. This field encodes the specific condition for the signal.

The high-order bit of the identity field (bit 15 of the condition value) is the customer definition flag. It distinguishes condition identification values for VSI-supplied software (bit set to 0) and non-VSI-supplied software (bit set to 1).

Condition values defined for application use must always have bit 15 set to 1 in order to avoid conflict with VSI-defined values.

A condition value is a BLOCK data structure (see *Section 11.10.3, "BLOCK Structures"*). The following declarations can be used to describe this structure:

```
FIELD
    CONDIT_FIELDS =
        SET
        STS$V_SEVERITY = [0,0,3,0],  ! Severity field
        STS$V_SUCCESS  = [0,0,1,0],  ! Success field
        STS$V_COND_ID  = [0,3,13,0], ! Identity field
        STS$V_CODE     = [0,3,12,0], ! Code for condition only
        STS$V_CUST_DEF = [0,15,1,0]  ! Customer definition flag
        TES;
MACRO
    CONDITION_VALUE = BLOCK[1] FIELD(CONDIT_FIELDS) %;
```

The following literal-declaration can be used to declare names for the codes used for the severity field of a condition value:

```
LITERAL
    STS$K_WARNING = 0,   ! Warning
    STS$K_SUCCESS = 1,   ! Successful Completion
    STS$K_ERROR   = 2,   ! Error
```

```
    STS$K_INFO    = 3,    ! Information
    STS$K_SEVERE  = 4;    ! Severe Error
```

Observe that these codes are chosen so that testing of the low order bit of the severity field will distinguish a successful condition (low bit equal to 1) from an unsuccessful condition (low bit equal to 0).

In the above declarations, the names used are the same as the names used in BLISS–32 (see *Section 17.6.1.2, "Condition Values in BLISS–32"*), which are based on names used in the VMS operating system.

As an aid to creating a condition value, the following keyword-macro-declaration is useful:

```
KEYWORDMACRO
    STS$VALUE(
        SEVERITY = STS$K_SEVERE,    ! default is severe error
        CODE,                       ! no default
        CUST_DEF = 1^15 )=          ! default is user definition
            (SEVERITY AND 7) OR
            (CODE AND %O'7777')^3 OR
            IF CUST_DEF NEQ 0
                THEN 1^15
                ELSE 0)
        %;
```

Comparing two condition values to determine if they represent the same condition must exclude the severity field. The following macro is useful for this purpose:

```
MACRO
    STS$MATCH(A,B) =
        (((A) AND %O'177770') EQL ((B) AND %O'177770')) %;
```

The macro returns true if two given condition values are equal and false otherwise.

The CHF-defined condition value needed in order to test for an unwind signal is provided as a global literal value. The following declaration can be used to declare the name of this literal:

```
EXTERNAL LITERAL
    SS$UNW;
```

17.6.1.2. Condition Values in BLISS–32

In BLISS–32, a condition value is a single fullword value that is encoded with three primary fields (proceeding from low-order to high-order): a severity field of 3 bits, an identity field of 25 bits, and a field of 4 bits that is reserved for system use.

The identity field is itself divided into two major fields: the message number field and the facility code field.

The 13 low-order bits of the identity field (bits 3 through 15 of the condition value) are the message number field. This field identifies the specific condition for the signal. The high-order bit (bit 15) distinguishes system wide codes (bit set to 0) that are common to all software (including user programs) and facility, specific (component) codes (bit set to 1).

The 12 high-order bits of the identity field (bits 16 through 27 of the condition value) are the facility code. This field identifies the specific software component in which the signal is generated. The high-

order bit (bit 27) distinguishes VSI-supplied software facilities (bit set to 0) and non-VSI-supplied facilities (bit set to 1).

Condition values defined for application use must always have both bits 15 and 27 set to 1 in order to avoid conflict with VSI-defined values. Application programs can use systemwide message number values provided they are used as defined for the VMS system.

A condition value is a BLOCK data structure (see *Section 11.9.3, "Semantics"*). The following declarations can be used to describe this structure:

```
FIELD
    CONDIT_FIELDS =
        SET
        STS$V_SEVERITY  = [0,0,3,0],   ! Severity field
        STS$V_SUCCESS   = [0,0,1,0],   ! Success field
                                       ! (subfield of severity)
        STS$V_COND_ID   = [0,3,25,0],  ! Identity field
        STS$V_MSG_NO    = [0,3,13,0],  ! Message number field
        STS$V_FAC_SP    = [0,15,1,0],  ! Facility-specific flag
        STS$V_CODE      = [0,3,12,0],  ! Code for condition only
        STS$V_FAC_NO    = [0,16,12,0], ! Facility code
        STS$V_CUST_DEF  = [0,27,1,0]   ! Customer definition flag
        TES;
MACRO
    CONDITION_VALUE = BLOCK[1] FIELD(CONDIT_FIELDS) %;
```

The following literal-declaration can be used to declare names for the codes used for the severity field of a condition value:

```
LITERAL
    STS$K_WARNING = 0,   ! Warning
    STS$K_SUCCESS = 1,   ! Successful Completion
    STS$K_ERROR   = 2,   ! Error
    STS$K_INFO    = 3,   ! Information
    STS$K_SEVERE  = 4;   ! Severe Error
```

Observe that these codes are chosen so that testing of the loworder bit of the severity field will distinguish a successful condition (low bit equal to 1) from an unsuccessful condition (low bit equal to 0).

As an aid to creating a condition value, the following keyword-macro-declaration is useful:

```
KEYWORDMACRO
    STS$VALUE (
        SEVERITY = STS$K_SEVERE,  ! default is severe error
        CODE,                     ! no default
        FAC_SP = 1^15,            ! default is facility specific
        FAC_NO = 0,               ! arbitrary default
        CUST_DEF = 1^27) =        ! default is user definition
            (SEVERITY AND 7) OR
            (CODE AND (1^13-1))^3 OR
            (IF FAC_SP NEQ 0
                THEN 1^15
                ELSE 0) OR
            (FAC_NO AND (1^12-1))^16 OR
            (IF CUST_DEF NEQ 0
                THEN 1^27
                ELSE 0)
```

```
        %;
```

Comparing two condition values to determine if they represent the same condition takes several steps. The following macro serves this purpose:

```
MACRO
    STS$MATCH(A,B)=
        BEGIN
        LOCAL
            QQQQA: CONDITION_VALUE,
            QQQQB: CONDITION_VALUE;
        QQQQA = (A);
        QQQQB = (B);
        IF NOT (.QQQQA[STS$V_FAC_SP] OR .QQQQB[STS$V_FAC_SP])
        THEN
            .QQQQA[STS$V_CODE] EQL .QQQQB[STS$V_CODE]
        ELSE
            .QQQQA[STS$V_COND_ID] EQL .QQQQB[STS$V_COND_ID]
        END %;
```

This macro returns true if two given condition values are equal and false otherwise.

The CHF-defined condition value needed in order to test for an unwind signal is provided as a global literal value. The following declaration can be used to declare the name of this literal:

```
EXTERNAL LITERAL
    SS$_UNWIND;
```

17.6.1.3. Condition Values in BLISS–36

In BLISS–36, a condition value is a single fullword value that is encoded with three primary fields (proceeding from loworder to highorder): a severity field of 3 bits, an identity field of 29 bits, and a field of 4 bits that is reserved for future use.

Note that, in the following descriptions, bit positions are expressed in accordance with the BLISS bit-numbering convention: bit 0 is the low-order or rightmost bit and bit 35 is the high-order or leftmost bit.

The identity field is itself divided into two major fields: the message number field and the facility code field.

The 15 low-order bits of the identity field (bits 3 through 17 of the condition value) are the message number field. This field identifies the specific condition for the signal. Message numbers with the high-order bit (bit 17) clear are reserved for VSI-supplied software.

The 14 high-order bits of the identity field (bits 18 through 31 of the condition value) are the facility code. This field identifies the specific software component in which the signal is generated. The high-order bit (bit 31) distinguishes VSI-supplied software facilities (bit set to 0) and non-VSI-supplied facilities (bit set to 1).

Condition values defined for application use must always have both bits 17 and 31 set to 1 in order to avoid conflict with VSI-defined values.

The four high-order bits (bits 32 through 35) are reserved for future use and should be set to zero.

The following declarations can be used to access the various fields of the BLISS– 36 condition value:

```
FIELD
    CONDIT_FIELDS =
```

```
        SET
        STS$V_SEVERITY = [0,0,3,0],   ! Severity field
        STS$V_SUCCESS  = [0,0,1,0],   ! Success field
                                      ! (subfield of severity)
        STS$V_COND_ID  = [0,3,29,0],  ! Identity field
        STS$V_MSG_NO   = [0,3,15,0],  ! Message number field
        STS$V_FAC_SP   = [0,17,1,0],  ! Facility specific flag
        STS$V_CODE     = [0,3,14,0],  ! Code for condition only
        STS$V_FAC_NO   = [0,18,14,0], ! Facility code
        STS$V_CUST_DEF = [0,31,1,0]   ! Customer definition flag
        TES;
MACRO
    CONDITION_VALUE = BLOCK[1] FIELD(CONDIT_FIELDS) %;
```

The following literal-declaration can be used to declare names for the codes used for the severity field of a condition value:

```
LITERAL
    STS$K_WARNING = 0,   ! Warning
    STS$K_SUCCESS = 1,   ! Successful Completion
    STS$K_ERROR   = 2,   ! Error
    STS$K_INFO    = 3,   ! Information
    STS$K_SEVERE  = 4;   ! Severe Error
```

Observe that these codes are chosen so that testing of the low order bit of the severity field will distinguish a successful condition (low bit equal to 1) from an unsuccessful condition (low bit equal to 0).

As an aid to creating a condition value, the following keyword-macro-declaration is useful:

```
KEYWORDMACRO
    STS$VALUE (
        SEVERITY = STS$K_SEVERE,      ! default is severe error
        CODE,                         ! no default
        FAC_SP = 1^17,                ! default is facility specific
        FAC_NO = 0,                   ! arbitrary default
        CUST_DEF = 1^31) =            ! default is user definition
            (SEVERITY AND $O'7') OR
            (CODE AND %O'37777')^3 OR
            (IF FAC_SP NEQ 0
                THEN 1^17
                ELSE 0) OR
            (FAC_NO AND %O'37777')^18 OR
            (IF CUST_DEF NEQ 0
                THEN 1^31
                ELSE 0)
        %;
```

Comparing two condition values to determine if they represent the same condition takes several steps. The following macro is useful for this purpose:

```
MACRO
    STS$MATCH(A,B)=
        BEGIN
        LOCAL
            QQQQA: CONDITION_VALUE,
            QQQQB: CONDITION_VALUE;
        QQQQA = (A);
```

```
        QQQQB = (B);
        IF NOT (.QQQQA[STS$V_FAC_SP] OR .QQQQB[STS$V_FAC_SP])
        THEN
            .QQQQA[STS$V_CODE] EQL .QQQQB[STS$V_CODE]
        ELSE
            .QQQQA[STS$V_COND_ID] EQL .QQQQB[STS$V_COND_ID]
        END %;
```

The macro returns true if two given condition values are equal and false otherwise.

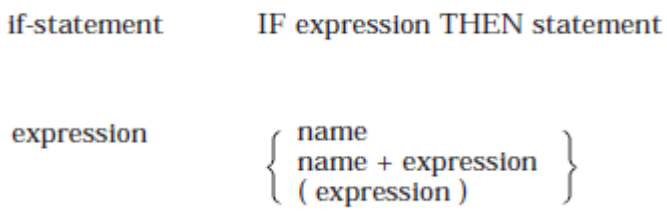
The CHF-defined condition value needed in order to test for an unwind signal is provided as a global literal value. The following declaration can be used to declare the name of this literal:

```
EXTERNAL LITERAL
    SS$UNW;
```

17.6.2. A Recursive-Descent Parser

A recursive-descent parser is a parser in which there is generally a one-to-one correspondence between the syntactic rules of the language and routines that parse constructs of the language. Each routine is designed to process one syntactic name and calls other routines to parse non-literal parts of the syntactic rule. The BLISS language is an example of a language that is suitable for this kind of parsing technique.

To begin this example, assume that the following two syntactic rules are part of a language to be parsed.



Further, assume that a routine named READ_LEX is available that reads the input for the parser, identifies the next lexeme, and assigns a code for the kind of lexeme to a data segment named LEXTYPE. This data segment must be declared with the VOLATILE attribute because, as will be seen later, its contents may be changed by a handler routine. The following names of lexical codes are used in the example:

Name of Code	Usage
LEX_IF	Keyword IF
LEX_THEN	Keyword THEN
LEX_NAME	A name
LEX_PLUS	Plus operator "+"
LEX_LPAREN	Left parenthesis "("
LEX_RPAREN	Right parenthesis ")"

The actual values for the codes are not important so long as they are distinct.

A routine to parse an if-statement can be written as follows:

```
ROUTINE SIF: NOVALUE =
```

```
    BEGIN
    READ_LEX();
    SEXPRESSION();
    IF .LEXTYPE NEQ LEX_THEN
    THEN
        BEGIN
        ERROR ('Missing THEN');
        RETURN
        END;
    READ_LEX();
    SSTATEMENT();
    END;
```

In this routine, the IF lexeme is recognized by some other parse routine, which then calls SIF. SIF calls READ_LEX to get the next lexeme in the input stream and then calls SEXPRESSION to parse an expression. When SEXPRESSION returns, the code for the first lexeme not accepted as part of an expression is still contained in LEXTYPE. Next SIF determines whether that lexeme is the keyword THEN. If not, an error is reported and SIF returns. Otherwise, READ_LEX is again called to get a new lexeme, SSTATEMENT is called to parse a statement, and SIF returns.

The routine SIF illustrates the close correspondence between the syntactic rule for the if-statement and the code that performs the parsing.

The code to parse an expression is more complicated, but is based on the same kind of correspondence. However, the name of the routine given next, which does the parsing for an expression, is SEXPRESSION1 instead of SEXPRESSION. The reason for this is discussed later. The code is as follows:

```
LITERAL
    EXP_ERROR = STS$VALUE(CODE = 1);

ROUTINE SEXPRESSION1: NOVALUE =
    BEGIN
    SELECTONE .LEXTYPE OF
        SET
        [LEX_LPAREN]:
            BEGIN
            READ_LEX();
            SEXPRESSION1();
            READ_LEX();
            IF .LEXTYPE NEQ LEX_RPAREN
            THEN
                BEGIN
                ERROR('Missing ")"') ;
                SIGNAL(EXP_ERROR)
                END;
            END;
        [LEX_NAME]:
            BEGIN
            READ_LEX();
            IF .LEXTYPE EQL LEX_PLUS
            THEN
                BEGIN
                READ_LEX();
                SEXPRESSION1();
                END;
            END;
```

```
        [OTHERWISE]:
            ERROR('Missing expression');
        TES;
    END;
```

An important aspect of this routine is that it recursively calls itself.

Consider what might happen if SEXPRESSION1 has recursed several levels when an error is detected. This would happen, for example, for the following invalid input for an expression:

```
(+Y+((Z(+Q))
       ^
```

The left parenthesis marked by the circumflex (^) is the point of error (a left parenthesis where there should be a right parenthesis). At this point SEXPRESSION1 has called itself three times. The problem is how to proceed after the error in a reasonable way. One simple strategy is to stop expression parsing, discard any subsequent lexemes that could be part of an expression, and then return to the routine that called for expression parsing in the first place.

A means to do this using condition handling is shown in the following pair of routines. The first routine, SEXPRESSION, is the establisher routine. The only purpose of SEXPRESSION is to establish the second routine, SEXP_ERROR, as a handler and then call SEXPRESSION1 to do the actual expression parsing. The routines are written as follows:

```
ROUTINE SEXPRESSION: NOVALUE =
    BEGIN
    ENABLE SEXP_ERROR;
    SEXPRESSION1();
    END;

ROUTINE SEXP_ERROR(SIG, MECH, ENAB) =
    BEGIN
    MAP
        SIG: REF VECTOR;
    BIND
        COND = SIG[1]: CONDITION_VALUE;

    ! Resignal all but EXP_ERROR, ignore unwind

    IF NOT STS$MATCH(.COND, EXP_ERROR)
    THEN RETURN 0;

    ! Skip all lexemes that can be part of an expression,
    ! Stop on any other lexeme.
    WHILE
        (SELECTONE .LEXTYPE OF
        SET
        [LEX_LPAREN, LEX_RPAREN, LEX_NAME, LEX_PLUS]: 1;
        [OTHERWISE]: 0;
        TES)
    DO
        READ_LEX();
    SETUNWIND();
    RETURN 0
    END;
```

The coding for SEXP_ERROR follows the template for condition handlers given in *Section 17.4.2, "Parameters"*, but is simplified because not all of the parameters are used. The coding also assumes the

declarations given in *Section 17.6.1, "Accessing and Defining Condition Values"* for accessing condition values.

If SEXPRESSION1 calls SIGNAL, then CHF skips over all of the calls to SEXPRESSION1 since no handler is established, and calls SEXP_ERROR.

SEXP_ERROR first tests whether the condition value is the one for an expression error. If not, then resignaling is requested. The same coding also causes an unwind signal to be ignored. It is valid in this case not to assign a return value for the establisher routine in the mechanism vector during unwinding because the establisher routine, SEXPRESSION, does not return a value. If the condition value does indicate an expression error, then the WHILE loop causes lexemes that could be part of the erroneous expression to be read and ignored. Recall that calling READ_LEX changes the contents of LEXTYPE. Because this change results from execution of a handler routine, LEXTYPE must be declared with the VOLATILE attribute. Finally, SETUNWIND is called to cause all of the calls to SEXPRESSION1 and the call to SEXPRESSION to be terminated.

17.6.3. Performance Measurement

In some cases condition handling is convenient for conducting certain kinds of performance measurement. This is particularly true when the analysis to be performed involves the dynamic calling relationship between routines.

For example, suppose the desired information is the relative number of times that a certain routine, say R, is called directly or indirectly by each of two other routines, say C1 and C2. You can accomplish this as follows:

1. Modify routine R to call SIGNAL at some appropriate point in its execution.
2. Modify routines C1 and C2 to establish handlers, say C1H and C2H.
3. Code C1H and C2H to increment counters each time a signal is received from R and then request continuation.
4. Execute the modified program to collect the frequency data and analyze the results.

It may also be prudent to modify the main routine to have a handler for the signal from R as well. This handler will be called if R signals when C1 or C2 is not in the stack of executing routine calls.

Observe that with this arrangement, if C1 calls C2 calls R, then the handler for C2 will be the one called.

It is, of course, possible to get the same frequency data by modifying the routines to set and test various counters and flags directly. But, in cases such as this one, condition handling may well be simpler and more convenient.

17.6.4. Target Operating Systems and Condition Handling

Target operating system support and use of condition handling is discussed briefly in the following sections.

17.6.4.1. PDP–11 Operating Systems

In BLISS–16, PDP–11 operating systems generally do not support condition handling as described in this manual, nor do they use condition handling in their internal operation. Condition handling for BLISS–16 is supported by the CHF in the BLISS–16 Run-Time Library.

17.6.4.2. The VMS Operating System

In BLISS–32, condition handling is directly supported by the condition handling facilities of the VMS operating system. The VMS system uses condition handling in several ways to achieve modular software components that can be flexibly used.

Condition handling plays a central role in reporting error messages. All error conditions are signaled using condition values and additional parameters that encode the error message to be reported. When DCL starts a user's program, it establishes its own handler, termed the catchall handler, in a stack frame prior to the stack frame for the main routine. Consequently, the catchall handler will be called for any signals that are not handled by the user's program.

The catchall handler is programmed to interpret the system's condition values and output the appropriate error messages. In addition, the catchall handler interprets the severity field as follows: If severe error is given, then the user program image is terminated; otherwise, the handler returns to CHF requesting continuation. Observe that if the signal was generated using SIGNAL_STOP, the severity will necessarily be severe error (see *Section 17.3.2, "Explicit Signals"* and *Section 17.4.3.2, "Resignaling"*).

This design provides considerable flexibility in adapting system software to various applications. On the one hand, a program that does not establish any handlers will receive standard system error messages. On the other hand, a program can establish a handler that will modify some or all of the system condition values in order to provide messages that are more appropriate to particular groups of users. For example, in a database inquiry application used by nontechnical users, a condition value for a subtle disk allocation problem can be replaced by a condition value for a message such as "System malfunction. Please call computer operations for assistance."

The VMS system provides *exception vectors* that provide a means to establish handlers that will be called before CHF begins searching the stack of routine calls for handlers and for certain cases where CHF encounters an invalid stack frame. The VMS Debugger module uses an exception vector to establish a handler to intercept signals for analysis and program testing purposes.

In certain special cases, the FORTRAN Run-Time Library establishes a handler between the command processor catchall handler and the user's main program to deal with various conditions specific to itself.

When reading the VMS manuals concerning condition handling, observe that the VMS software calls a handler with two parameters, the signal vector and mechanism vector, rather than three parameters as described in *Section 17.4.2, "Parameters"*. The BLISS system itself provides the enable vector parameter in addition to the two provided directly by VMS.

17.6.4.3. TOPS–10 and TOPS–20 Operating Systems

In BLISS–36, the TOPS–10 and TOPS–20 operating systems generally do not support condition handling as described in this manual, nor do they use condition handling in their internal operation. Condition handling for BLISS–36 is supported by the CHF software in the BLISS–36 Run-time Library.

Chapter 18. Special Features

The preceding chapters describe declarations for the names of data, structures, routines, conditions, bound values, lexical functions, and macros. This chapter describes the remaining declarations of BLISS. These declarations make use of the general declaration mechanism of BLISS for some rather specialized purposes. They are as follows:

- The psect-declaration, which specifies the required properties of the program sections used in a program
- The switch-declaration, which permits the specification of compiler switches for any block of a program
- The built-in-declaration, which makes available certain names that are predefined but not predeclared
- The label-declaration, which is used in connection with the exit-expressions
- The undeclare-declaration, which cancels the effect of any other kind of declaration for a given name

18.1. Psect-Declarations

The psect-declaration allows you to inform the linker about the storage characteristics required for different sections of your program, and allows you to group various kinds of object code in an efficient manner.

You can, for example, request that a given program section be write-protected (which it normally might not be), or request that a given section be allocated in the same memory space as a section by the same name from another module. Also on some target systems you can request that a given section be shareable by several different processes.

Most of the program section characteristics, called psect-attributes, are very target-system specific. Therefore, the psect-declaration is in general not transportable, although it can be used transportably in a limited fashion.

A psect-declaration can be used to allow a BLISS program to share data with a program written in another language. In the VAX environment, for example, another use of the psect-declaration allows a set of modules to share a workspace whose size is determined by the linker, based on the needs of the particular set of modules present.

A psect-declaration can also be used to provide a second level of control over program organization. The first level of control is specified by the division of a program into modules. A second level of control is sometimes necessary if the division into modules (and the default program sections, where supplied) does not by itself provide the best organization of storage for efficient execution or debugging.

Examples of psect-declarations are given in the following block:

```
OWN
    A,
    B;
PSECT OWN = ALPHA(NOWRITE);
OWN
    C,
    D,
    E;
PSECT OWN = BETA(EXECUTE);
OWN F: VECTOR[10];
```

The data segments for the OWN variables A and B are allocated in the default program section for the storage-class OWN. The data segments for C, D, and E are allocated in the program section ALPHA, which cannot be written into. The data segment for F is allocated in the program section BETA, which can be executed.

BLISS is unusual if not unique among higher-level languages in providing the kind of storage-allocation control permitted by the psect-declaration.

18.1.1. Syntax

```
psect-declaration       PSECT psect-item , . . . ;
psect-item              storage-class = psect-name

                          { (psect-attribute , . . . ) }
                          { nothing                    }

storage-class           { OWN        }
                        { GLOBAL     }
                        { PLIT       }
                        { CODE       }
                        { NODEFAULT  }

psect-name              name

psect-attribute         { WRITE | NOWRITE                  }
                        { EXECUTE | NOEXECUTE              }
                        { OVERLAY | CONCATENATE            }
                        { b16-psect-attribute   <= 16 Only }
                        { b32-psect-attribute   <= 32 Only }
                        { b36-psect-attribute   <= 36 Only }

16 Only =>

    b16-psect-attribute {LOCAL | GLOBAL}

32 Only =>

    b32-psect-attribute { READ | NOREAD                    }
                        { SHARE | NOSHARE                  }
                        { PIC | NOPIC                      }
                        { LOCAL | GLOBAL                   }
                        { VECTOR                           }
                        { alignment-attribute              }
                        { addressing-mode-attribute        }

36 Only =>

    b36-psect-attribute { READ | NOREAD                    }
                        { ORIGIN(address-expression)       }

address-expression      compile-time-constant-expression
```

The alignment-attribute is described in *Section 9.5, "The Alignment-Attribute – BLISS–16/32 Only"* and the addressing-mode-attribute is described in *Section 9.13, "The Addressing-Mode-Attribute – BLISS–16/32 Only"*.

18.1.2. Restrictions

In the definition of the psect-attribute, most attributes are given in mutually exclusive pairs: WRITE and NOWRITE, OVERLAY and CONCATENATE, and so on. You cannot use both members of such a pair in declaring a single psect-name. The alignment-attribute, the addressing-mode-attribute, and the ORIGIN attribute are not members of such pairs.

All declarations of a given psect-name in a program must provide the same set of psect-attributes for the name. This restriction is applied after any missing attributes have been supplied by the default rules.

BLISS–32 ONLY

The value of the boundary expression in an alignment-attribute for a program section must be in the range 0 through 9.

The value of that boundary expression must not be exceeded by the value of the boundary expression in an alignment-attribute for any data segment that is allocated in the program section.

BLISS–36 ONLY

A psect-name must be unique among all other psect-names within its first six characters.

If a declaration of a psect-name other than LOW or $HIGH$ appears in a module, the first (or only) such declaration must appear before any data- or routine-declarations (other than the external or forward forms), and before any expression containing a PLIT. That is, it must appear before the first declaration that causes storage to be allocated or object code to be generated.

The value of the address-expression in the ORIGIN attribute must be in the range 0 to (2**18)–1 inclusive.

18.1.3. Defaults

BLISS–16 ONLY

The following psect-declaration is assumed to appear in a block that surrounds each module:

```
PSECT
    OWN    = $OWN$     (WRITE,NOEXECUTE,CONCATENATE,LOCAL),
    GLOBAL = $GLOBAL$  (WRITE,NOEXECUTE,CONCATENATE,LOCAL),
    PLIT   = $PLIT$    (NOWRITE,NOEXECUTE,CONCATENATE,LOCAL),
    CODE   = $CODE$    (NOWRITE,EXECUTE,CONCATENATE,LOCAL);
```

This declaration provides a default program section name for each of the four storage-classes. The psect-attributes used are the default psect-attributes that are given in the following paragraph.

If a psect-item contains a parenthesized list of psect-attributes, then any missing attributes are filled in by default. The defaults are as follows:

Attribute	Default

WRITE | NOWRITE	WRITE
EXECUTE | NOEXECUTE	NOEXECUTE (EXECUTE for CODE)
OVERLAY | CONCATENATE	CONCATENATE
LOCAL | GLOBAL	LOCAL

BLISS–32 ONLY

The following psect-declaration is assumed to appear in a block that surrounds each module:

```
PSECT
    OWN     = $OWN$          (READ,WRITE,NOEXECUTE,NOSHARE,
                              NOPIC,CONCATENATE,LOCAL,ALIGN(2),
                              ADDRESSING_MODE(WORD_RELATIVE)),
    GLOBAL = $GLOBAL$        (READ,WRITE,NOEXECUTE,NOSHARE,
                              NOPIC,CONCATENATE,LOCAL,ALIGN(2),
                              ADDRESSING_MODE(WORD_RELATIVE)),
    PLIT    = $PLIT$         (READ,NOWRITE,NOEXECUTE,NOSHARE,
                              NOPIC,CONCATENATE,LOCAL,ALIGN(2),
                              ADDRESSING_MODE(WORD_RELATIVE)),
    CODE    = $CODE$         (READ,NOWRITE,EXECUTE,NOSHARE,
                              NOPIC,CONCATENATE,LOCAL,ALIGN(2),
                              ADDRESSING_MODE(WORD_RELATIVE));
```

This declaration provides a default program section name for each of the four storage-classes. The psect-attributes used are the default psect-attributes that are given in the following paragraph.

If a psect-item contains a parenthesized list of psect-attributes, then any missing attributes are filled in by default. The defaults are as follows:

Attribute	**Default**
READ | NOREAD	READ
WRITE | NOWRITE	WRITE (NOWRITE for PLIT or CODE)
EXECUTE | NOEXECUTE	NOEXECUTE (EXECUTE for CODE)
SHARE | NOSHARE	NOSHARE
PIC | NOPIC	NOPIC
OVERLAY | CONCATENATE	CONCATENATE
LOCAL | GLOBAL	LOCAL
alignment-attribute	ALIGN(2)
addressing-mode-attribute	ADDRESSING_MODE(WORD_RELATIVE)

BLISS–36 ONLY

The following psect-declaration is assumed to appear in a block that surrounds each module:

```
PSECT
    OWN     = $LOW$          (READ,WRITE,EXECUTE,CONCATENATE,
                              ORIGIN(0)),
    GLOBAL = $LOW$           (READ,WRITE,EXECUTE,CONCATENATE,
```

```
                          ORIGIN(0)),
    PLIT    = $HIGH$     (READ,NOWRITE,EXECUTE,CONCATENATE,
                          ORIGIN(%O'400000')),
    CODE    = $HIGH$     (READ,NOWRITE,EXECUTE,CONCATENATE,
                          ORIGIN(%O'400000'));
```

This declaration provides a default program-section name for each of the four storage-classes. The psect-attributes used are the default psect-attributes that are given in the following paragraph.

If a psect-item contains a parenthesized list of psect-attributes, then any missing attributes are filled in by default. The defaults are as follows:

Attribute	**Default**
READ | NOREAD	READ
WRITE | NOWRITE	WRITE (NOWRITE for PLIT or CODE)
EXECUTE | NOEXECUTE	EXECUTE
OVERLAY | CONCATENATE	CONCATENATE

There is no default for the ORIGIN attribute: if it is not specified, then the corresponding program-section origin must be specified at link time (**/SET** qualifier of the **LINK** command). Further, there is no default for the address-expression of this attribute.

If a psect-item does not contain a parenthesized list of psect-attributes and if a previous declaration of the psect-name is given in the module, then the psect-attributes are taken from the first declaration of the same psect-name.

18.1.4. Semantics

NODEFAULT is a special storage-class which allows the declaration of a psect without overriding current defaults for OWN, GLOBAL, PLIT, or CODE data; thus, the current defaults need not be either known or restored. For example, the following declarations allow a longword to be shared between BLISS–32 and VAX PL/I:

```
PSECT
    NODEFAULT = PLI_DATA(ADDRESSING_MODE(ABSOLUTE),OVERLAY,READ,WRITE);
OWN
    PLI_DATA : PSECT(PLI_DATA);
```

With the last declaration, PL/I will expect global and external symbols to be declared in an overlaid psect of the same name; moreover, it has not been necessary to redeclare the defaults.

In the following sections, the semantics of the psect-declaration are given in four parts. First, the storage-classes are described. Next, the program section attributes are given. Then, psect-names and their scope are discussed. Finally, the interpretation of a psect-declaration is given.

18.1.4.1. Storage-Classes

The storage-class in a psect-item determines the kind of data that is allocated in the corresponding program section. The following list indicates the declarations or primaries that are associated with each storage-class.

Declaration or Primary	**Storage-Class**

OWN declarations	OWN
GLOBAL declarations	GLOBAL
PLITs	PLIT
ROUTINE and GLOBAL ROUTINE declarations	CODE

In other words, any data segments allocated by the compiler in processing OWN declarations are allocated in program sections declared for the storage-class OWN; any data segments allocated in processing GLOBAL data declarations are allocated in program sections for the storage-class GLOBAL; and so on.

18.1.4.2. Psect-Attributes

The following attributes of a program section provide information to the linker about the way the program section should be allocated in storage:

READ | NOREAD (32/36 Only)
WRITE | NOWRITE
EXECUTE | NOEXECUTE
OVERLAY | CONCATENATE
SHARE | NOSHARE (32 Only)
PIC | NOPIC (32 Only)
LOCAL | GLOBAL (16/32 Only)
VECTOR (32 Only)
ALIGN(boundary) (32 Only)
ADDRESSING_MODE(mode) (32 Only)
ORIGIN(address) (36 Only)

The READ, WRITE, and EXECUTE attributes determine which kinds of access to the program section are permitted. Based on these attributes, the linker establishes the hardware memory-management access control needed for the storage of the program section, assuming that a target system's hardware/software environment does in fact provide the required facilities. That is, for transportability purposes, attributes designated for a given system that have no effective meaning in another are allowed in the corresponding dialect because they will be ignored.

The OVERLAY attribute causes program sections that have the same name but come from different modules to be allocated in the same storage (like FORTRAN COMMON blocks, for example). The CONCATENATE attribute causes program sections with the same name from different modules to be allocated contiguously, each in its own storage.

BLISS–16/32 ONLY

The LOCAL and GLOBAL attributes provide indicators for the target-system linker, which uses them in the allocation and management of physical memory for a program. In BLISS–16, these indicators direct the construction of program overlays. In BLISS–32, these indicators direct the grouping of pages within a program image so as to optimize performance.

BLISS–32 ONLY

The SHARE attribute specifies that the program section can be accessed by more than one process.

The PIC (position-independent code) attribute indicates that the program section can be relocated without affecting its validity.

The alignment-attribute causes the storage for the program section to begin with a byte whose address ends with at least n zero bits, where n is the value of the boundary expression in the alignment-attribute. This attribute also causes the storage for the program section to be extended, if necessary, with unused bytes until its last byte is just before a byte whose address ends with at least n zero bits. Thus, for example, an ALIGN(1) attribute causes a program section to begin and end at word boundaries, an ALIGN(2) at longword boundaries, and so on. The alignment-attribute is further described in *Section 9.5, "The Alignment-Attribute – BLISS–16/32 Only"*.

The addressing-mode-attribute determines the addressing mode for each data segment allocated in the program section. The significance of the addressing mode is given in *Section 9.13, "The Addressing-Mode-Attribute – BLISS–16/32 Only"*.

The VECTOR psect-attribute causes generation of an indication to the linker that the program section contains entry-point vector information for a VMS privileged shared image, used in the construction of shared run-time libraries. This attribute is analogous to the VEC attribute in VAX MACRO.

BLISS–36 ONLY

The ORIGIN attribute specifies the machine address at which a program section is to start. For example, ORIGIN(%O'400000') will cause the corresponding program section to start at the standard high-segment beginning address, 400000 octal. Note that the use of this attribute can result in unallocated storage left between two program sections, or in overlapping program sections. Proper use of this attribute must be guided by familiarity with the linker for the target system in question.

A complete understanding of the program-section attributes requires knowledge of the way storage is or can be laid out by the linker. Information on the allocation of storage can be found in the appropriate linker (or task-builder) reference manual for the target system. See also the appropriate BLISS user manual for additional information.

18.1.4.3. Psect-Names

A psect-name is interpreted by the linker and is, necessarily, global to a module. The first declaration of a given psect-name within a module serves two purposes. First, it establishes the name and defines the attributes for the program section associated with that name for the scope of the module. Second, the first declaration of a given name establishes the program section associated with that name as the current program section for the storage class in the scope in which it is declared. Thus, unless a NODEFAULT storage class is used to prevent an override of the default attributes, subsequent declarations of the psect-name will serve only the second purpose, which is to establish the current program section for a storage-class. All declarations of a particular psect-name within a module must be equivalent. Psect-declarations are equivalent if one of the following applies:

- The declarations are identical.
- The declarations have the same set of attributes after the missing attributes have been filled in by default.
- The second of the two declarations has no parenthesized list of attributes. In this case, the attributes from the first declaration apply to the second declaration.

18.1.4.4. Interpretation

Every use of the same psect-name in a program refers to the same program section. A psect-declaration not only states (or restates) the psect-attributes for a given program section, but also selects that program section for use within the scope of the declaration for a given storage-class.

18.1.5. Discussion

The simplest way to ensure that all declarations of a psect-name in a given module are equivalent is to use the simple form of a psect-declaration, in which no parenthesized list of attributes is given, for all psect-declarations except the first one. Consider the following program segment:

```
BEGIN
ROUTINE S=
    BEGIN
    PSECT OWN = ALPHA (NOWRITE);
    OWN S1;
    ...
    END
OWN A, B;
PSECT OWN = BETA;
OWN C;
...
PSECT OWN = ALPHA;
OWN D;
...
END
```

The first declaration of the psect-name ALPHA defines the name and establishes its attributes; in BLISS–32, for example, the code can be as follows:

```
READ, NOWRITE, NOEXECUTE, NOSHARE, NOPIC, CONCATENATE, LOCAL,
ALIGN(2), ADDRESSING_MODE(WORD_RELATIVE)
```

The NOWRITE attribute is given explicitly in the psect-declaration and the other attributes are determined by default. The subsequent declaration of the psect-name ALPHA does not have a parenthesized list of attributes; therefore, the list associated with the previous declaration is assumed. Note that giving these declarations in the opposite order results in an error.

Data and routines from different storage-classes can be allocated in the same program section by means of the appropriate psect-declarations. For example, suppose that all PLITs for a given module must be allocated in the same program section that is used for the object code for routines. Then the following declaration can be written in the outer block of the module:

```
PSECT
    PLIT = $CODE$;
```

This declaration overrides the default psect-declaration for the PLIT storage-class, which allocates PLITs in the program section named $PLIT$.

Suppose the following declaration appears in an inner block of the module in the previous paragraph:

```
PSECT
    PLIT = $PLIT$;
```

Within the block in which this declaration appear, PLITs are allocated in the default program section for PLITs, just as if the declaration mentioned in the preceding paragraph were not present.

18.2. Switches-Declarations

A switches-declaration allows you to give the compiler additional information about the desired interpretation of a block. In this way, each block can be given individual treatment by the compiler.

For example, a block that is still in the debugging process can have a switches-declaration that causes the compiler to provide listings, error messages, and macro expansion traces for that block. Or, a block in an inner loop can have a switches-declaration that causes the compiler to perform special optimizations.

An example of a switches-declaration is given in the following block:

```
BEGIN
...
    BEGIN
    SWITCHES NOERRS;
    ...
    END;
...
END
```

The inner block has a switches-declaration that specifies that no warning or error messages are to be displayed for that block.

Some switch-items, such as ADDRESSING_MODE, simply set attribute defaults for the remainder of the block, and thus have only an indirect effect, that is, through other declarations later in the block that take those defaults.

In general, the actions or interpretations requested by a switches-declaration take effect only after the occurrence of the declaration (from the viewpoint of code generation). Therefore, in the normal case where the effect is desired throughout the block in question, the correct positioning of the switches-declaration is at the very beginning of the block (that is, prior to any code-producing declaration).

18.2.1. Syntax

```
switches-declaration    SWITCHES switch-item , . . . ;

switch-item             { on-off-switch-item    }
                        { special-switch-item   }

on-off-switch-item      { ERRS | NOERRS                 }
                        { OPTIMIZE | NOOPTIMIZE         }
                        { SAFE | NOSAFE                 }
                        { UNAMES | NOUNAMES             }
                        { ZIP | NOZIP                   }

special-switch-item     { ADDRESSING_MODE               }   ⇐ 32 Only
                        {   (mode-spec , . . . )        }
                        { LANGUAGE (language-list)      }
                        { LINKAGE (linkage-name)        }
                        { LIST (list-option , . . . )   }
                        { STRUCTURE                     }
                        {   ({structure-attribute})     }
                        {    {nothing}                  }

language-list           { COMMON                        }
                        { language-name , . . .         }
                        { nothing                       }

language-name           {BLISS16 | BLISS32 | BLISS36}

linkage-name            name

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

32 Only ⇒
mode-spec               { EXTERNAL = mode               }
                        { NONEXTERNAL = mode            }
```

32 Only ⇒

mode { GENERAL | ABSOLUTE | LONG_RELATIVE | WORD_RELATIVE }

The structure-attribute is defined in *Section 11.4, "Structure-Attributes and Storage Allocation"*.

18.2.2. Restrictions

An ADDRESSING_MODE switch can have no more than one EXTERNAL mode-spec and no more than one NONEXTERNAL mode-spec (BLISS–32 only).

The linkage-name in a LINKAGE switch must either be explicitly declared as a linkage-name in a containing block or must be a predeclared linkage-name.

The structure-name in the structure-attribute of a STRUCTURE switch must either be explicitly declared as a structure-name in a containing block or must be a predefined structure-name.

18.2.3. Defaults

If a switch-item is not specified, the setting established by the compilation command specification, by the module-head or by a switches-declaration in an outer block is assumed.

If a null language-list appears in a LANGUAGE switch (that is, LANGUAGE ()), the single language-name corresponding to the compiler in use is assumed. This implies that no transportability checking is to be performed within the scope of the containing block.

If the keyword COMMON appears in the language-list of the LANGUAGE switch, it is equivalent to the explicit specification of all three language-names.

18.2.4. Semantics

The switch-items specify actions to be taken by the compiler in processing a block.

In addition to the following description, additional discussion of the compiler actions for these switches can be found in the BLISS user manual for the appropriate compiler.

18.2.4.1. On-Off-Switch-Items

Each on-off-switch-item has a negation, which consists of the switch-item prefixed by the characters NO. The negation of a switch-item indicates that the associated action should not be taken. The action associated with each switch-item is given in the following list:

Switch-Item	Action
ERRS	Print warnings and error messages from the compiler on the terminal.
OPTIMIZE	Perform optimization across mark points.
SAFE	Ignore computed addresses in doing optimization.
UNAMES	Generate unique names for OWN variables, nonglobal ROUTINE names, and labels when producing a listing that is to be assembled.

ZIP	Optimize time at the expense of space.

18.2.4.2. Special-Switch-Items

The special-switch-items provide additional information about the block being compiled. The action associated with each special-switch-item is given in the following list:

Switch-Item	**Action**
ADDRESSING_MODE (mode-spec, . . .) (BLISS–32 Only)	Establish the given addressing modes as the addressing-mode defaults for subsequent declarations in the current block. An EXTERNAL mode-spec supplies the default for EXTERNAL and EXTERNAL ROUTINE declarations. A NONEXTERNAL mode-spec supplies the default for FORWARD, FORWARD ROUTINE, and PSECT declarations. This default is ineffective unless a program section is declared within the block. The addressing-mode attribute is described in *Section 9.13, "The Addressing-Mode-Attribute – BLISS–16/32 Only"*.
LANGUAGE (language-list)	Establish the given list of language-names for the remainder of the current block. Perform transportability checking, if applicable, for the combination of dialects specified or implied in the list. See *Section 18.2.5, "Discussion"* and *Appendix C, "Transportability Checking"* for further information.
LINKAGE (linkage-name)	Establish the given linkage-name as the linkage-name default for the remainder of the current block. This linkage-name is used as the linkage-attribute of any subsequent routine declaration in the current block that does not specify a linkage-attribute.
LIST (list-option, . . .)	Establish the given list-options for the output listing of the remainder of the current block. The list-options are described in the following subsection.
STRUCTURE (structure-attribute)	Establish the given structure-attribute as the default structure-attribute to be used in subsequent default-structure-references within the current block (see *Section 11.4, "Structure-Attributes and Storage Allocation"* and *Section 11.8, "Default-Structure-References"*). If the given structure-attribute is null, then all subsequent default-structure-references in the block are invalid.

18.2.4.3. List-Options

The output listing produced as a result of a BLISS compilation can contain the following separate parts:

Source listing
Macro expansions and traces
Library usage traces
Object code listing

The LIST switch-item controls the parts of the output listing to be produced according to the settings specified by the list-options. The first two list-options, SOURCE and REQUIRE, operate on a special counter, the *source listing counter*. The counter is initially set to 1, and source text is listed when, and only when, the value of the counter is greater than zero. Thus the SOURCE and REQUIRE list-options control the listing of the source text from files specified in the compilation command and by REQUIRE declarations.

The action associated with each list-option is given in the following list.

List-Option	**Action**
SOURCE	Increments the source listing counter. NOSOURCE decrements the source listing counter.
REQUIRE	Causes the source listing counter to be left unchanged when a file specified by a REQUIRE declaration is opened or closed. NOREQUIRE causes the source listing counter to be decremented when a file specified by a REQUIRE declaration is opened, and incremented when the file is closed.
EXPAND	List the lexeme stream that is the result of each macro expansion.
TRACE	Trace the expansion of macros, printing each lexeme stream produced during the expansion and the final lexeme stream produced as a result of the expansion.
LIBRARY	Trace the usage of names whose declarations are obtained from library binary files.
OBJECT	List the object code. The format of the listing is determined by the settings of the following four switches.
ASSEMBLY	List the object code instructions in a form suitable for assembly.
SYMBOLIC	List the object code instructions in a form suitable for interpretation by the programmer. This format uses source program symbols wherever possible in the object code instructions.
BINARY	List the binary text of the object code.
COMMENTARY	List commentary produced by the compiler concerning the object code generated. At present, commentary is limited to a line-number cross-reference.

18.2.5. Discussion

The LANGUAGE switch is an aid in the development of transportable programs. As a module-switch, it declares your intention to compile the module under several different compilers, for use on the corresponding target systems. It requests that the compiler analyze the module from the standpoint of transportability. For example, with two compiler names specified in the LANGUAGE switch, both compilers will check for and report the occurrence of certain machine-sensitive language features that may pose problems when the module is processed by the other compiler.

Used in a SWITCHES declaration, this switch essentially allows you to turn off transportability checking within the block immediately containing the declaration. For example, the need for this capability arises where a given block is not coded transportably, is inherently machine- or system-dependent, and must be modified for each target system. The specific language constructs that are checked for a given set of target systems are described in *Appendix C, "Transportability Checking"*. Briefly, these constructs fall into the following categories:

- All syntactic features that are not common to the target set. For example, if all three target systems are specified, then the occurrence of any dialect-specific feature is reported.
- Most syntactic features that, although common to the target set, are likely *in certain forms* to cause transportability problems (for example, string-literals used as primary expressions).
- Certain dialect-sensitive elements that may occur in otherwise valid constructs; for example, field-selector values that are compile-time constant expressions are checked at compile time for conformance to the restrictions imposed by the most restrictive target system.

In general, the checks performed in response to the LANGUAGE switch alert you to language features that most often require special attention when transporting programs. Such checking cannot, however, identify or resolve all of the problems that may be encountered. In particular, the functional equivalence of a program in several different environments cannot be assured (at compile time) in all cases, even though the program compiles successfully in each environment.

Each BLISS user manual contains a section on "Transportability Guidelines". A study of this section and frequent, parallel compilations of the module to be transported are strongly recommended.

18.3. Built-In-Declarations

Certain names are predefined in BLISS. Some of the predefined names are predeclared, so that they can be used without being declared explicitly; an example is the name ABS, which is the name of the absolute-value function. Other predefined names are not predeclared, but must, instead, be declared BUILTIN before they can be used.

The classification of a given predefined name as predeclared or built-in is part of the BLISS language definition; it is given in *Appendix A, "Predefined Identifiers"*. Names that are frequently used and that apply to all dialects of BLISS are predeclared. Names that are predefined only in certain dialects of BLISS are built in. In particular, all names of machine-specific-functions are built in; these are listed in *Appendix D, "Built-In Functions"*.

18.3.1. Syntax

built-in-declaration	BUILTIN built-in-name , . . . ;
built-in-name	name

18.3.2. Restrictions

Each name in a built-in-declaration must be listed in *Appendix A, "Predefined Identifiers"* under the classification "built-in name".

A built-in-declaration containing a predefined register-name (see *Section 10.7.4, "Pragmatics"*) or a predefined name of a linkage-function (see *Section 13.6, "Linkage-Functions"*) must be contained in a routine-declaration.

18.3.3. Semantics

A built-in-declaration informs the compiler that the names listed are used as built-in-names in the current block.

The full definition of each built-in-name is given elsewhere in the definition of BLISS. For example, in BLISS–16 the built-in-name PC is a register-name and is defined in *Section 10.7.4, "Pragmatics"*. For another example, the built-in-name BICPSW is a VAX machine-specific-function name and is defined in the *BLISS–32 User Manual*.

18.4. Label-Declarations

The use of labels is very restricted in BLISS. Labels are used only to identify a block so that a LEAVE expression can be used to terminate the evaluation of the block. When a label is used, it must be declared by a label-declaration.

18.4.1. Syntax

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

18.4.2. Semantics

A label declaration informs the compiler that the names listed are used as labels in the current block.

The use of labels is discussed in connection with exit-expressions in *Section 6.6, "Exit-Expressions"*.

18.5. Undeclare-Declarations

An undeclare-declaration is used to limit the scope of a declaration. An undeclare-declaration in an inner block prevents references to names declared in outer blocks. An undeclare-declaration may also be used in a library source file to prevent a name from being entered into the precompiled library binary file (see *Section 16.6, "Library-Declarations"*).

An example of an undeclare-declaration follows:

```
BEGIN
OWN A,B,C;
...
BEGIN
UNDECLARE A,C;
...
```

```
END
END
```

In the inner block, the name B designates the OWN variable declared in the outer block, but the names A and C have no meaning.

18.5.1. Syntax

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

18.5.2. Semantics

An undeclare-declaration informs the compiler that each undeclared-name in the list has no declared meaning for the scope of the current block.

A name that is undeclared may be subsequently declared for some other use within the scope of the declaration.

A name that is undeclared at the end of a library compilation is not entered in the library binary file produced by the compiler.

18.5.3. Pragmatics

In order to redeclare a macro-name it must be "quoted" using the lexical function %QUOTE (see *Section 15.5.14, "Quote-Functions"*). Effectively this inhibits expansion of the macro-name at the point of redeclaration. For example, to undeclare the name ZYX declared as a macro-name elsewhere in the same module, the following form of declaration is required:

```
UNDECLARE %QUOTE ZYX
```

This requirement applies to any other redeclaration of a macro-name as well.

Chapter 19. Modules and Programs

This chapter describes modules and programs. No new functional capability is introduced here; instead, the way in which a program interfaces with the compiler in particular and the target system in general is described.

This chapter has four sections. The first section describes modules in a general way. The second section completes the description of modules by defining the module-switches. The third section describes the predefined names, which provide one form of connection between programs and the system. The fourth section describes programs.

19.1. Modules

The module is the compilation unit of BLISS. Each module is complete for purposes of compilation. However, a module is usually incomplete for purposes of execution because it often depends on information supplied by the other modules with which it is linked to form a program. The use of GLOBAL and EXTERNAL declarations allows these points of communication to be identified so that their resolution can occur at link time.

The division of a program into modules helps define the fundamental organization of the program. Declarations that have some property in common can be grouped into a single module. For example, if two routine-declarations are always used together, then grouping them in a module ensures that they are allocated together. For another example, if some declarations are subject to change when a new version of the program is produced, then grouping them together in a module makes it possible to change the program by only recompiling a single module.

An example of a module is as follows:

```
MODULE COMPOOL (IDENT = '000015') =
BEGIN
    GLOBAL LITERAL
    BUFSIZ = 226,
    PAGESIZ = 132,
FACTOR = 33: SIGNED(9);
GLOBAL BIND X = PLIT (0,1,2,3,4,5,6,7,8,9,10,11,12)
                : VECTOR[13];
END
ELUDOM
```

This module contains the constant declarations that are used in other modules of the program.

Another example of a module follows:

```
MODULE STK (IDENT = '000001') =
BEGIN
OWN STK: VECTOR[1000];
OWN STKPTR: INITIAL(0);
EXTERNAL ROUTINE
    STKERR1,
    STKERR2;
GLOBAL ROUTINE PUSH(X): NOVALUE =
```

```
    BEGIN
    IF .STKPTR GEQ 1000 THEN STKERR1();
    STKPTR = .STKPTR + 1;
    STK[.STKPTR] = .X;
    END;
GLOBAL ROUTINE POP(X): NOVALUE =
    BEGIN
    IF .STKPTR LSS 0 THEN STKERR2();
    .X = .STK[.STKPTR];
    STKPTR = .STKPTR-1;
    END;
END
ELUDOM
```

This module contains both data-declarations and routine-declarations.

19.1.1. Syntax

module	MODULE module-head = module-body ELUDOM
module-head	module-name { (module-switch , . . .) / nothing }
module-name	name
module-body	block

19.1.2. Restrictions

A module-body's outermost level can contain only declarations; that is, it must be a sequence of declarations within a BEGIN-END or parenthesis pair. Some of these declarations can be routine-declarations, and these define the actions that can be performed by the module.

Some declarations must not be given at the outermost level of a module, namely, declarations of temporary data segments and linkage-functions. These are local-declarations (*Section 10.5, "Local-Declarations"*), stacklocal-declarations (*Section 10.6, "Stacklocal-Declarations"*), register- declarations (*Section 10.7, "Register-Declarations"*), and built-in-declarations (*Section 18.3, "Built-In-Declarations"*) that give any of the predefined register names (*Section 10.7.4, "Pragmatics"*) or any of the names of linkage-functions (*Section 13.6, "Linkage-Functions"*).

19.1.3. Semantics

A module provides the compiler with three items:

- The *module-name*, which is used in some contexts by the compiler to identify the object code for the module.
- The *module-switches*, which select various options offered by the compiler.
- The *module-body*, which is translated by the compiler from BLISS into an object code file.

19.2. Module-Switches

Module-switches allow you to control some aspects of the compiler's treatment of the module. Because you know the module's stage of development and its intended use, you can use switches to cause additional operations to be performed and to suppress other operations. Consider the development of a typical module from syntax checking through debugging into production. At the beginning, the module is as follows:

```
MODULE M1 (IDENT = '0001', NOCODE, LIST(TRACE),
           LANGUAGE(BLISS16,BLISS32)) =
BEGIN
...
END
ELUDOM
```

In this example, the module-switches direct the compiler to perform only a syntax check (NOCODE) and to trace the expansion of macros (LIST(TRACE)). The LANGUAGE switch signifies the intent to compile the module with both the BLISS–16 and the BLISS–32 compilers. It requests that the compiler currently in use check for and report the appearance of dialect-sensitive language features that might cause problems in transporting the module across the specified systems. Switches that are not given explicitly are determined by the default rules. For example, the switch ERRS is assumed by default and therefore the compiler prints warnings and error messages on your terminal.

Later, when the module is being debugged, the module's switches are changed to the following:

```
MODULE M1 (IDENT = '0005', DEBUG, NOOPTIMIZE) =
BEGIN
...
END
ELUDOM
```

In this version, the module-switches direct the compiler to prepare the symbol table and the linkages required for use by a debugger and to omit certain kinds of optimization by the compiler of the generated object code (NOOPTIMIZE). When the module is ready for production, the switches are changed as follows:

```
MODULE M1 (IDENT = '0203') =
BEGIN
...
END
ELUDOM
```

In this version, all the switches except the identification switch are omitted because the default rules are oriented toward a production module.

19.2.1. Syntax

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

on-off-switch

{ CODE | NOCODE
DEBUG | NODEBUG
ERRS | NOERRS
OPTIMIZE | NOOPTIMIZE
SAFE | NOSAFE
UNAMES | NOUNAMES
ZIP | NOZIP }

special-switch

{ common-switch
BLISS–16-switch
BLISS–32-switch
BLISS–36-switch }

⇐ 16 Only
⇐ 32 Only
⇐ 36 Only

common-switch

{ IDENT = quoted-string
LANGUAGE (language-list , . . .)
LINKAGE (linkage-name)
LIST (list-option , . . .)
STRUCTURE
({ structure-attribute })
{nothing }
MAIN = routine-name
OPTLEVEL = { 0 | 1 | 2 | 3 }
VERSION = quoted-string }

language-list

{ COMMON
language-name , . . .
nothing }

language-name

{BLISS16 | BLISS32 | BLISS36 }

list-option

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

{ linkage-name
routine-name }

name

16 Only ⇒
bliss–16-switch

{ ADDRESSING_MODE (mode-16)
ENVIRONMENT (environ-16-option , . . .) }

mode-16

{ABSOLUTE | RELATIVE }

environ-16-option

{ EIS | NOEIS | LSI11 | T11 | PIC | ODT }

32 Only ⇒

bliss–32-switch

ADDRESSING_MODE (mode-spec , . . .)

mode-spec

{ EXTERNAL = mode-32 | NONEXTERNAL = mode-32 }

mode-32

{ GENERAL | ABSOLUTE | LONG_RELATIVE | WORD_RELATIVE }

36 Only ⇒

bliss–36-switch

ADDRESSING_MODE (mode-36)
ENTRY (global-name , . . .)
ENVIRONMENT (environ-36-option , . . .)
OTS = quoted-string
OTS_LINKAGE = linkage-name

mode-36

{INDIRECT | NOINDIRECT }

environ-36-option

{ cpu-option | monitor-option | ots-option | stack-option }

cpu-option

{ KA10 | KI10 | KL10 | KS10 | EXTENDED }

monitor-option

{ TOPS10 | TOPS20 }

ots-option

{ BLISS10_OTS | BLISS36C_OTS }

stack-option

STACK = segment-name

{ global-name | linkage-name | segment-name }

name

The structure-attribute is defined in *Section 11.4, "Structure-Attributes and Storage Allocation"*.

19.2.2. Restrictions

The MAIN switch must appear once and only once in a program.

The routine-name specified in the MAIN switch must be declared in a ROUTINE or GLOBAL ROUTINE declaration in the same module.

The VERSION switch can appear only in a module that also contains the MAIN switch.

The name specified in the structure-attribute of a STRUCTURE switch must be a predeclared structure-name.

BLISS–36 ONLY

Each name specified in the ENTRY switch must be declared GLOBAL, GLOBAL ROUTINE, GLOBAL BIND, GLOBAL BIND ROUTINE, or GLOBAL LITERAL in the same module.

The ots-option of the ENVIRONMENT switch must not appear together with either the OTS switch or the OTS_LINKAGE switch.

The stack-option of the ENVIRONMENT switch may appear only in a module that also contains the MAIN switch.

The quoted-string given in the VERSION switch must conform to the TOPS–10/20 version-number format:

```
ooo a (oooooo) -o
```

where *o* represents an octal digit and *a* represents an alphabetic character. Leading zeros are not required.

The linkage-name in the OTS_LINKAGE switch must either be predeclared or appear in a linkage-declaration preceding the first routine-declaration in the module. The named linkage-definition must not specify register parameter- locations or global-registers.

19.2.3. Defaults

If a setting for an on-off-switch is not given, the following default settings are used:

On-Off-Switch Default	**Action**
CODE	Generate object code.
NODEBUG	Do not build table and linkages for the debugger.
ERRS	Print compiler diagnostic messages on terminal.
OPTIMIZE	Optimize across mark points.
SAFE	Ignore computed addresses in performing optimization.
NOUNAMES	Do not generate unique names.
NOZIP	Do not optimize time at the expense of space.

If a setting for a special-switch is not given, the following defaults are assumed:

Special-Switch Default	Action
ADDRESSING_MODE(RELATIVE)	(BLISS–16 Only) Use the relative addressing mode for all generated instructions.
ADDRESSING_MODE(EXTERNAL = WORD_RELATIVE, NONEXTERNAL = WORD_RELATIVE)	(BLISS–32 Only) Use the short/relative form of address encoding as the ultimate addressing-mode default.
ADDRESSING_MODE(NOINDIRECT)	(BLISS–36 Only) Do not use the indirect addressing mode for any generated instructions.
ENVIRONMENT(EIS)	(BLISS–16 Only) Produce the object module using instructions from the Extended Instruction Set (ASH, ASHC, DIV, MUL, SOB, SXT) wherever appropriate.
LANGUAGE(%BLISS16(BLISS16) %BLISS32(BLISS32) %BLISS36(BLISS36))	The module is intended for compilation only by the compiler currently in use, and no transportability checking is to be performed. See *Section 16.2.4, "Predeclared Macros"* for a description of the predeclared macros shown above.
LINKAGE(BLISS) (BLISS–16/32 Only) LINKAGE(BLISS36C) (BLISS–36 Only)	Use the predefined linkage BLISS in BLISS– 16 and -32, or the predefined linkage BLISS36C in BLISS–36, for any routine that does not specify a linkage-attribute.
LIST(SOURCE, NOREQUIRE, NOEXPAND, NOTRACE, NOLIBRARY, OBJECT, NOASSEMBLY, SYMBOLIC, BINARY, COMMENTARY)	List the source text, but not the text contributed by files specified in require- declarations. Do not list macro expansions or traces. Do not list library usage traces. List the object code instructions using symbolic names, the binary text, and commentary produced by the compiler.
STRUCTURE()	That is, the default structure-attribute is empty, and default-structure-references are invalid (see *Section 11.8, "Default-Structure-References"*).
OPTLEVEL=2	Perform all optimizations that can be invoked without making any special assumptions about the program.

The BLISS–36 ENVIRONMENT switch defaults, except for the ots-option and stack-option, are established when a given BLISS–36 compiler is generated. See the *BLISS–36 User's Guide* for details.

The default for the ots-option is BLISS36C_OTS. This implies the standard BLISS36C Object Time System file name for a given target environment, and implies the standard BLISS36C linkage for generating OTS routine calls.

If the stack-option is not specified in a module that contains the MAIN switch, a 2048-word stack is established by default.

The defaults for the OTS and OTS_LINKAGE switches are, respectively, the standard OTS filename and the standard OTS linkage established by the (explicit or default) ENVIRONMENT switch ots-option. More specifically, the OTS_ LINKAGE default can be either BLISS36C or BLISS10, depending upon the ots-option setting.

If a null language-list appears in a LANGUAGE switch (that is, LANGUAGE()), the single language-name corresponding to the compiler in use is assumed. (This is equivalent to the default for the entire LANGUAGE switch, as described above.)

19.2.4. Semantics

The module-switches inform the compiler to take or suppress an action. The actions associated with the special-switches and on-off-switches are described in the following sections.

19.2.4.1. Special-Switches

The special-switches ADDRESSING_MODE (in BLISS–32), LANGUAGE, LINKAGE, LIST, and STRUCTURE can be used in a switches-declaration as well as in a module-head; those switches are described in *Section 18.2, "Switches-Declarations"*. See *Appendix C, "Transportability Checking"* for further information on the LANGUAGE switch and transportability checking.

The special-switches that can be used only as module-switches are defined as follows:

Special-Switch	**Action**
ADDRESSING_MODE(mode-16) (BLISS–16 Only)	Generate instructions using absolute or relative addressing mode as indicated.
ADDRESSING_MODE(mode-36) (BLISS–36 Only)	Generate instructions using indirect or noindirect addressing mode as indicated.
ENTRY(name , . . .) (BLISS–36 Only)	Produce an object-module record that contains the specified global (entry) names, for use by the linker when forming a library of object modules.
ENVIRONMENT(environ-16-option) (BLISS–16 Only)	EIS: Generate object code employing instructions from the PDP–11 Extended Instruction Set. NOEIS: Generate object code employing only the instructions available to all PDP–11 models. LSI11: Generate object code employing only the instructions available to the LSI–11 processor. T11: Generate object code employing only the instructions available to the T11 processor. PIC: Generate position-independent code. ODT: Facilitate debugging with ODT.
ENVIRONMENT(environ-36-options) (BLISS–36 Only)	Cpu-option: Specifies the processor model of the target system for which code is to be generated. Monitor-option: Specifies the operating system of the target system for which code is to be generated. Ots-option: Specifies which of the standard object-time systems is to be used (at link-time) to satisfy outstanding external references, and implies the corresponding standard linkage to be used for OTS

	calls (which may differ from the default linkage for non-OTS calls). Stack-option: Specifies the name of an OWN or GLOBAL data-segment declared in the same (main) module to be used as the control stack for the program, in place of a default compiler-generated segment.
IDENT = 'xxx' (BLISS–16/32 Only)	Include the quoted-string as an identification in the object module generated from the compilation of the module. See the appropriate BLISS user manual for any applicable restrictions.
MAIN = routine-name	Save the routine-name. Program execution will begin with a routine-call on the routine designated by this routine-name.
OPTLEVEL = level	Use the value of level as a guide for the kind of optimizations performed, as follows: 0 – Minimum Optimization 1 – Low Optimization 2 – Normal Optimization 3 – Maximum Optimization The level value 0 produces the most readable object code.
OTS = 'ots-file-spec' (BLISS–36 Only)	Use the specified object-module library file when searching for object-time- system routines instead of the standard OTS file implied by the ots-option (see ENVIRONMENT). Note that LINK-20 requires that the quoted file-spec conform to the TOPS–10 style (DEV:[PPN]filnam).
OTS_LINKAGE = linkage-name (BLISS–36 Only)	Use the named linkage-definition when generating calls to the object-time-system identified in the OTS switch.
VERSION = 'version-number' (BLISS–36 Only)	Include the quoted-string as an identification in the executable image of the program generated by linking the main module containing this switch.

BLISS–32 ONLY

The quoted-string given with the IDENT special-switch is printed by the linker in the map it produces as a result of linking the modules of a program. This quoted-string usually contains an identifier that is used to determine which version of an object module is present in a program.

BLISS–36 ONLY

The quoted-string given with the VERSION special-switch is placed in the "version number" location of the executable image produced as a result of linking the modules of a program. Note that the module containing the VERSION switch must also contain the MAIN switch. This quoted-string must contain a conventional version number that is used to identify the version level of a program.

19.2.4.2. On-Off-Switches

The on-off-switches ERRS, OPTIMIZE, SAFE, UNAMES, and ZIP can be used in a switches-declaration as well as in a module-head; these switches are described in *Section 18.2, "Switches-Declarations"*. The on-off-switches that can be used only as module-switches are defined as follows:

On-Off-Switch	Action
CODE	Generate the object code for the module.
DEBUG	Build the symbol table and the linkages required for use of the debugger.

Each of these switches has a negation, formed by prefixing the switch name with NO. The negated switch means that the indicated action should not be taken.

19.3. Predefined Names

Some names have a predefined, specific meaning that is part of the definition of BLISS. For example, ABS is the name of the absolute value function, and VECTOR is the name of a predefined vector structure.

There are two kinds of predefined names: *predeclared* and *built-in*. The predeclared names can be used without any declaration; indeed, a predeclared name must not be declared wherever it is used in its predefined sense. On the other hand, a built-in name must be declared BUILTIN wherever it is used in its predefined sense.

It is important to note that predefined names are not reserved. A predefined name can be declared for some user purpose (for example, as the name of a data segment or a macro or a routine). Within the scope of such a declaration, the predefined meaning of the name is lost; but if that meaning is not required, no damage is done.

The names that are predefined in the versions of BLISS that are described in this manual are listed in the following paragraphs. Additional predefined words will be added to BLISS as the language grows.

Predeclared standard-function-names

The following names are predeclared as standard-function-names:

- SIGN, ABS
- MAX, MAXU, MAXA
- MIN, MINU, MINA
- %REF

The description for each of these standard-function names is given in *Section 5.2, "Executable-Functions"*.

Built-in register-names

The predefined register-names must be declared BUILTIN wherever they are used as such. The register-names that are predefined for each dialect are described in *Section 10.7.4, "Pragmatics"*.

Predeclared structure-names

The following names are predeclared as names for predefined structures:

- BITVECTOR
- BLOCK
- BLOCKVECTOR
- VECTOR

The structure-declaration for each of these structure-names is given in *Section 11.10, "Predeclared Structures"*.

Predeclared linkage-names

The following names are predeclared as linkage- names:

- BLISS (16/32 Only)
- FORTRAN (16/32 Only)
- FORTRAN_FUNC
- FORTRAN_SUB
- BLISS36C (36 Only)
- BLISS10 (36 Only)

The description of these linkage-names is given in *Section 13.5, "Common Predeclared Linkage-Names"*.

Built-in linkage-functions

The following predefined names of linkage-functions must be declared BUILTIN wherever they are used as such:

- ACTUALCOUNT
- ACTUALPARAMETER
- ARGPTR
- NULLPARAMETER (16/32 Only)

The description of these linkage-functions is given in *Section 13.6, "Linkage-Functions"*.

Predeclared condition-handling-functions

The following names are predeclared as names of condition-handling-functions:

- SETUNWIND
- SIGNAL
- SIGNAL_STOP

The description of these condition-handling-functions is given in *Chapter 17, "Condition Handling"*.

Predeclared macro-names

The following names are predeclared as macro-names:

- %BLISS16
- %BLISS32
- %BLISS36

The description for each of these macro-names is given in *Section 16.2.4, "Predeclared Macros"*.

Predeclared supplementary-function-names

The following names are predeclared as supplementary-function-names:

- CH$ALLOCATION, CH$SIZE
- CH$PTR, CH$PLUS, CH$DIFF
- CH$RCHAR, CH$A_RCHAR, CH$RCHAR_A
- CH$WCHAR, CH$A_WCHAR, CH$WCHAR_A
- CH$MOVE, CH$FILL, CH$COPY
- CH$COMPARE
- CH$EQL, CH$NEQ, CH$LSS, CH$LEQ, CH$GTR, CH$GEQ
- CH$FIND_CH, CH$FIND_NOT_CH, CH$FIND_SUB, CH$FAIL
- CH$TRANSTABLE, CH$TRANSLATE

All of these are names of functions in the character-handling package, which is described in *Chapter 20, "Character-Handling Functions"*.

Built-in machine-specific-function names

Each BLISS dialect provides a set of predefined machine-specific-function names that must be individually declared BUILTIN wherever they are used as such. The machine-specific-functions defined for each dialect are described in the appropriate BLISS user's guide. The function names (for all dialects) are included in the listing of predefined identifiers given in *Appendix A, "Predefined Identifiers"* of this manual.

19.4. Programs

A program is made up of object modules that have been linked together to form a single executable unit. The object modules that make up the program are produced as a result of the translation of a source module by one of the translators in the system. For example, the BLISS compiler translates BLISS modules into object modules and the FORTRAN compiler translates FORTRAN programs into object modules. Each translator produces an object module with a uniform set of indicators for the linker. The linker uses these indicators to allocate the modules and resolve points of communication among them.

Consider a program that inputs values, sorts them, and then outputs the same values in sorted order. This program could consist of a FORTRAN program to do input/output and the following BLISS modules:

```
MODULE TREESORT (IDENT = '0002')
BEGIN
ROUTINE EXCHANGE(F1,F2) =
    ...;
GLOBAL ROUTINE TREESORT(F1,F2) =
    ...;
END
ELUDOM

..

MODULE PROCESS (IDENT = '0002',MAIN = PROCESS) =
BEGIN
EXTERNAL ROUTINE
    INPUT: FORTRAN,
    OUTPUT: FORTRAN,
    TREESORT;
ROUTINE PROCESS =
    BEGIN
    PSECT OWN = ALPHA;
    OWN A: VECTOR[100];
    INPUT(A);
    TREESORT(A,100);
    OUTPUT(A)
    END;
END
ELUDOM
```

The linker links the two object modules produced by a BLISS compiler and the FORTRAN object module produced by the FORTRAN compiler to form a single unit. Then, execution begins at the specified point. In this case, execution begins with the routine PROCESS.

Chapter 20. Character-Handling Functions

A major part of computing is devoted to *character handling*; that is, the manipulation of sequences of characters. Character handling is required for the interpretation of user commands, for the preparation of output listings, for the management of symbol tables, for the editing of text, and for the maintenance of files.

This chapter describes the BLISS functions that are designed for character handling. Some of these functions perform a basic operation, such as allocating storage for a character sequence, or creating a pointer that can move back and forth through a character sequence, or writing (or reading) a character at a given position in a character sequence. Other functions perform an operation on an entire character sequence, such as moving, copying, comparing, or searching the sequence.

The functions described in this chapter are part of the set of *supplementary- functions* that was introduced in *Section 5.2, "Executable-Functions"*. A call on one of these functions usually does not produce a subroutine call; instead, it is compiled into a few hardware instructions that are specially designed for character handling. These functions provide a way of using these hardware instructions without causing a program to be machine-dependent. A program that uses these functions correctly (and that does not have machine dependence elsewhere) can be transported without change to another BLISS target system.

The first section of this chapter presents the concepts that are necessary for a discussion of character handling. The second section defines the character handling functions.

20.1. Fundamental Concepts

A discussion of the fundamental concepts of character handling follows. First character data is described, and then the operations that are applied to character data are summarized.

20.1.1. Character Sequence Data

A **character code** is a sequence of bits that represents a character. Usually the ASCII encoding of characters is used in BLISS. However, as long as a program makes consistent use of a given character encoding, it does not matter what that encoding is.

A **character position** is the storage for a single character code. For a given implementation of BLISS, the size of a character position is determined by two factors: the requirements of the character set and the organization of the computer memory. A program can be written in a way that does not depend on the specific character size used by a specific implementation.

A **character position sequence** is a portion of storage that is used for one or more character positions. Such a sequence has a first and last position. For each position except the first, there is a previous position, and for each position except the last, there is a next position.

A **character data segment** is a character position sequence that is allocated as a single portion of storage. In the simpler applications of character handling, it is possible to treat each character data segment as a separate unit, allocated in the same way other data segments are. In more advanced applications, a single character position sequence may extend across several data segments and may be reorganized as program execution proceeds.

A **character pointer** is a value that designates a character position. Sometimes a character pointer is set to the first character position of a sequence and remains there, providing access to the entire sequence. In other cases, a character pointer is used to scan back and forth in a sequence, selecting one position after another. A character pointer occupies a fullword. It can be moved from one fullword to another or can be passed as a parameter of a routine, like any other fullword value. However, a character pointer can be correctly interpreted only by a character-handling function. For example, a character pointer must be advanced by the CH$PLUS function, not by the plus sign (+) operator.

A **null pointer** is a returned value that indicates the absence of a valid character pointer. A null pointer results from the unsuccessful search for one or more characters within a sequence. The presence of a null pointer can only be tested for by a CH$FAIL function, and a null pointer must not be passed to any other character function.

The **length** of a character position sequence is the number of character positions in the sequence. The length of a sequence is not included as part of the sequence itself. In order to fully specify a character position sequence, both its length and a pointer to its first position must be given. Typically, the parameters of the character handling functions occur in pairs, a length followed by a pointer.

Character handling can be programmed on two levels. On the simpler level, all the data is divided into independent character data segments, and the segments are allocated in the usual way for OWN or LOCAL segments. In more advanced applications, data can be allocated dynamically, under program control.

20.1.2. Character Sequence Operations

The basic operations of character handling are summarized here. These operations are the allocating of storage, creating of a pointer, moving a pointer, fetching or storing a character code, and comparing of character sequences. A character data segment is allocated in a special way. Specifically, the amount of storage required is expressed in terms of character positions rather than longwords, words, or bytes.

A character pointer is created from a given data segment address. The data segment must be one that was allocated as a character sequence segment. The character pointer designates the first character position of the sequence.

A character pointer that designates a given character position is moved forward by changing it to designate the next character position of the sequence. Similarly, a character pointer is moved backward by changing it to designate the previous character position of the sequence. A character pointer should not be moved beyond the character data segment in which it originated unless you are quite sure what lies beyond that segment or you intend to move it back into the same segment before using it.

The contents of a character position must always be fetched or stored by means of a character pointer that designates the character position. In contrast, a character pointer can be fetched or stored like any other fullword value (by means of the fetch operator (.) or the assignment operator (=)).

Character sequences and character pointers must be compared only by means of the character handling functions designed for that purpose.

20.2. Functions

For the purpose of definition, the character handling functions are arranged in eight classes, as follows:

Allocation functions
Pointer functions
Character-reading functions

Character-writing functions
Sequence-writing functions
Sequence-comparing functions
Sequence-searching functions
Sequence-translating functions

Each class of functions is described in one of the following sections.

The name of each character handling function consists of the prefix CH$ followed by a mnemonic name; for example, CH$ALLOCATION is the name of the function that computes the storage that must be allocated for a sequence.

20.2.1. Allocation Functions

The allocation functions determine the amount of storage required for character data. The function CH$ALLOCATION returns the number of fullwords required for a given number of characters. The function CH$SIZE returns the number of bits required for a single character.

20.2.1.1. Definition

The allocation functions are defined as follows:

CH$ALLOCATION(*n*, *cs*)

Interpret *n* as an unsigned integer (the length of the allocated sequence). Interpret *cs* as an unsigned integer (the character size). Imagine a character position sequence composed of *n* character positions, each of which occupies *cs* bits. Return the number of fullwords that would be required for storage of such a character position sequence.

Default character size: The character-size parameter can be omitted; that is, the form CH $ALLOCATION(*n*) is permitted. In this case, the system default for the character size is used for *cs*. In BLISS–16 and BLISS–32 this default is 8; in BLISS–36, the default is 7.

CH$SIZE(*ptr*)

Interpret *ptr* as a pointer to a character position sequence. Return the character size for the sequence; that is, return the number of bits occupied by each character position of the sequence.

Default character size: The pointer parameter can be omitted; that is, the form CH$SIZE() is permitted. In this case, the system default for character size is returned.

The character size (*cs*) must be a compile-time constant expression.

The CH$ALLOCATION function is a compile-time constant expression if the length parameter (*n*) is a compile-time constant expression.

The CH$SIZE function is a compile-time constant expression if the pointer parameter (*ptr*) is omitted.

In BLISS–16 and BLISS–32, a function that specifies a character size other than 8 is invalid. Thus, the character size is a constant in BLISS–16 and BLISS–32. While the character size in BLISS–36 variable, with a range of 1 through 36 bits, any departure from the default 7-bit character size for ASCII encodings or the 6-bit character size for the SIXBIT encoding must be used with caution.

20.2.1.2. Examples

The CH$ALLOCATION function is normally used within the VECTOR attribute. An example of this usage follows:

```
OWN
    S3: VECTOR[CH$ALLOCATION(80)];
```

This declaration allocates a character data segment for S3 that is composed of 80 character positions.

The use of CH$ALLOCATION within the VECTOR attribute is a way of extending the BLISS language to handle character data without making major changes in the design of the language. Specifically, the use of the VECTOR attribute is a way of allocating storage for a character position sequence. It follows that storage allocated in this way should not be accessed as a vector, even though that is technically possible. Instead, the storage should always be accessed by the character-handling functions.

In fact, the combination of the VECTOR attribute with CH$ALLOCATION should be thought of as a single language construct, as in the following macro:

```
MACRO
    CH$SEQUENCE(N) = VECTOR[CH$ALLOCATION(N)] %;
```

Within the scope of this declaration, CH$SEQUENCE can be used as if it were a character-sequence attribute. For example, the declaration of S3 can be written as follows:

```
OWN
    S3: CH$SEQUENCE[80];
```

The CH$SEQUENCE macro just given is not a predeclared part of the BLISS language. It is given here as a suggested user-declared macro. If it is used in a program, then it must be explicitly declared in that program.

When the CH$ALLOCATION function is used in the VECTOR attribute (as is normally the case), the parameters of CH$ALLOCATION must be compile-time- constant-expressions. This restriction follows from the definition of the VECTOR attribute (given in *Section 11.4.1, "Syntax"*, which requires that an expression that is an actual parameter of the VECTOR attribute be a compile-time constant expression.

The declaration of S3, given above, satisfies this requirement because its length parameter is 80 and its character-size parameter is absent.

In advanced programming applications, CH$ALLOCATION is used with a nonconstant length. For example, in a program that performs dynamic allocation of storage for character sequences, CH $ALLOCATION is used to determine the amount of storage required.

20.2.2. Pointer Functions

The pointer functions create or manipulate character pointers. The CH$PTR function returns a character pointer that designates a character position. The CH$PLUS function creates a character pointer that is offset by a given number of character positions from another character pointer. The CH$DIFF function determines the offset between two given character pointers.

20.2.2.1. Definition

The pointer functions are defined as follows:

CH$PTR(*addr*, *i*, *cs*)

Interpret *addr* as the address of a data segment (the base address). Interpret *i* as a signed integer (the index). Interpret *cs* as an unsigned integer (the character size). Assume that the given segment is a

character position sequence that uses *cs* bits for each character position. Return a character pointer to the (*i*+1)th character position of the sequence contained in the segment at *addr*.

Default character size: The character-size parameter can be omitted; that is, the form CH $PTR(addr,i) is permitted. In this case, the system default is used for the character size. In BLISS–16 and BLISS–32, this default is 8; in BLISS–36, the default is 7.

Default index: When the character-size parameter is omitted, the index parameter can also be omitted; that is, the form CH$PTR(addr) is permitted.

In this case, the system default is used for the character size and zero is used for the index.

CH$PLUS(*ptr*, *i*)

Interpret *ptr* as a pointer into a character position sequence. Interpret *i* as a signed integer (the index). Suppose that *ptr* designates the kth character position of the given sequence. Return a pointer that designates the (*i*+k)th character position of the given sequence.

CH$DIFF(*ptr1*, *ptr2*)

Interpret *ptr1* and *ptr2* as character pointers of the same character size (bits per character) pointing into the same character position sequence. Suppose the pointers designate the n1th and n2th character positions, respectively, of the given sequence. Return (n1–n2).

The character size (*cs*) in a CH$PTR function must be a compile-time constant expression, and in BLISS–16 and BLISS–32 its value must be 8.

The CH$PTR function is a link-time constant expression if *addr* is a link-time constant expression and *i* and *cs* are, if given, each a compile-time constant expression.

In BLISS–16 and BLISS–32 a function that specifies a character size other than eight bits is not valid.

20.2.2.2. Examples

A character data segment is allocated with a name whose value is an address. Because a character position sequence must be accessed through a character pointer, some means for creating a pointer is required. The CH$PTR fills this need.

An example of the use of the CH$PTR function follows:

```
LITERAL
    BUFFSIZE = 80;
OWN
    QADDR: CH$SEQUENCE[BUFFSIZE],
    QBEGIN,
    QEND;
...
QBEGIN = CH$PTR(QADDR);
QEND   = CH$PTR(QADDR,BUFFSIZE-1);
```

The two assignments set the contents of QBEGIN and QEND to pointers to the first and last character positions of the segment QADDR. CH$SEQUENCE is a user-declared macro that was described in *Section 20.2.1.2, "Examples"*.

Given a pointer to a character position, the CH$PLUS function can produce a modified pointer that designates a character position that is a certain number of positions before or after the original position. For example:

```
LITERAL
    BUFFSIZE = 80;
OWN
    X: CH$SEQUENCE[BUFFSIZE],
    PTR1;
...
PTR1 = CH$PTR(X);
INCR I FROM 0 TO BUFFSIZE-1 DO
    BEGIN
    ...    ! Operation #1
    PTR1 = CH$PLUS(.PTR1,1);
    END;
```

This loop evaluates Operation #1 (which is not specified here) BUFFSIZE times. During each evaluation, PTR1 designates a different character position within X, starting at the first position and advancing by one position each time.

Given two pointers, the number of characters between them can be obtained by means of the CH$DIFF function. For example:

```
OWN
    M: CH$SEQUENCE[100];
    PTR1,
    PTR2,
    N;
...
PTR1 = CH$PTR(M,25);
PTR2 = CH$PTR(M,75);
...
N = CH$DIFF(.PTR2,.PTR1);
```

This program fragment sets N to 50, which is the offset of PTR2 relative to PTR1.

The CH$DIFF function is the only valid way to compare two character pointers. Suppose, for example, it is necessary to call the routine REX if the pointer contained in X is the same as the pointer contained in Y. This action can be programmed as follows:

```
IF CH$DIFF(.X,.Y) EQL 0 THEN REX();
```

20.2.3. Character-Reading Functions

Each of the character-reading functions returns a character code. Specifically, each function uses a given character pointer to locate a character position, and then fetches the character code that is contained in that character position. The functions operate on the given character pointer in different ways: CH $RCHAR does not change the pointer, CH$A_RCHAR advances the pointer by one character position before fetching a character code, and CH$RCHAR_A advances the pointer after fetching.

20.2.3.1. Definition

The character-reading functions are defined as follows:

CH$RCHAR(*ptr*)

> Interpret *ptr* as a character pointer. Fetch the contents of the character position that is designated by the character pointer. Return the fetched value.

CH$A_RCHAR(*addr*)

Interpret *addr* as the address of a character pointer. Advance the character pointer to the next character position and then fetch the contents of the character position designated by the character pointer. Return the fetched value.

CH$RCHAR_A(*addr*)

Interpret *addr* as the address of a character pointer. Fetch the contents of the character position designated by the character pointer and then advance the character pointer to the next character position. Return the fetched value.

Note that the parameter of CH$RCHAR is a character pointer, whereas the parameter of CH $A_RCHAR and CH$RCHAR_A is the address of a character pointer.

20.2.3.2. Examples

For some examples of these functions, consider the following program fragment:

```
CP = CH$PTR(UPLIT('ABCD'));          !Creates pointer to sequence.
CV1 = CH$RCHAR(.CP);                 !Sets CV1 to %C'A'.
CV2 = CH$A_RCHAR(CP);                !Sets CV2 to %C'B'.
CV3 = CH$RCHAR_A(CP);                !Sets CV3 to %C'B'.
CV4 = CH$RCHAR(.CP);                 !Sets CV4 to %C'C'.
```

20.2.4. Character-Writing Functions

Each of the character-writing functions stores a character code. Specifically, each function uses a given character pointer to locate a character position, and then stores a given character-code in that character position. Like the character- reading functions, these functions operate on the given character pointer in different ways: CH$WCHAR does not change the pointer, CH$A_WCHAR advances the pointer by one position before storing the character code, and CH$WCHAR_A advances the pointer after storing.

20.2.4.1. Definition

The character-writing functions are defined as follows:

CH$WCHAR(*c*, *ptr*)

Interpret *c* as a character code and interpret *ptr* as a character pointer. Store *c* in the character position designated by the character pointer. Do not return a value.

CH$A_WCHAR(*c*, *addr*)

Interpret *c* as a character code and interpret *addr* as the address of a character pointer. Advance the character pointer to the next character position, then store *c* in the character position designated by the character pointer. Do not return a value.

CH$WCHAR_A(*c*, *addr*)

Interpret *c* as a character code and interpret *addr* as the address of a character pointer. Store *c* in the character position designated by the character pointer, then advance the character pointer to the next character position. Do not return a value.

In each of these functions, *c* must be in a range suitable for use as a character code. Because none of these functions return a value, they must not be used in contexts that require a value. As with the

character-reading functions, the parameter of CH$WCHAR is a character pointer, whereas the parameter of CH$A_WCHAR and CH$WCHAR_A is the address of a character pointer.

20.2.4.2. Examples

An example of the use of these functions is the following program fragment:

```
OWN
    S4: CH$SEQUENCE[5],
    P: INITIAL(CH$PTR(S4));
...
CH$WCHAR(%C'P',.P);
INCR I FROM 1 TO 4 DO
    CH$A_WCHAR(%C'Q',P);
```

This example fills S4 up with 'PQQQQ'.

20.2.5. Sequence-Writing Functions

Each of the sequence-writing functions sets the contents of a character position sequence. The CH $MOVE function copies a specified number of characters from one character position sequence into another. The CH$FILL function sets all of the character positions of a sequence to a given character code; for example, it can initialize a sequence to all blanks. The CH$COPY function is relatively complex; it can copy several separate character sequences into a given character position sequence and then fill in any remaining positions with a given fill character. Thus a single CH$COPY function acts like a series of CH$MOVE functions followed by a CH$FILL function.

20.2.5.1. Definition

The sequence-writing functions are defined as follows:

CH$MOVE(*n*, *sptr*, *dptr*)

Interpret *n* as an unsigned integer (the length of both source and destination). Interpret *sptr* and *dptr* as pointers. Use these pointers to locate two character position sequences (the source and the destination, respectively).

Copy *n* characters from the source into the destination. That is, copy the contents of the first character position of the source into the first character position of the destination, copy the contents of the second character position of the source into the second character position of the destination, and so on, until n characters have been copied. Return a pointer to the (*n*+1)th character position of the destination.

CH$FILL(*fill*, *dn*, *dptr*)

Interpret *fill* as a character code. Interpret *dn* as an unsigned integer (the length of the destination). Interpret *dptr* as a character pointer. Use the pointer to locate the beginning position of a character position sequence (the destination).

Copy *fill* into the first *n* character positions of the destination. Return a pointer to the (*dn*+1)th character position of the destination.

CH$COPY(*sn1*, *sptr1*, *sn2*, *sptr2*, . . . , *fill*, *dn*, *dptr*)

Interpret *sn1, sn2,* . . . , and *dn* as unsigned integers (the lengths of the sources and the destination). Interpret *sptr1, sptr2,* . . . , and *dptr* as character pointers. Use *sptr1, sptr2,* . . . , and *dptr* to

locate the beginning positions of some character position sequences (the first source, the second source, . . . , and the destination, respectively). Interpret *fill* as a character code.

Copy *sn1* character codes from the first source into the first *sn1* character positions of the destination, copy *sn2* character codes from the second source into the next *sn2* character positions of the destination, and so on. If less than *dn* characters have been copied, copy the character code *fill* into the remaining character positions of the destination. Return a pointer to the (*dn*+1)th character position of the destination.

If the source lengths, *sn1, sn2,* and so on, are all compile-time constant expressions, then *sn1*+*sn2*+ . . . must not be greater than *dn*. If the lengths of the sources are not all compile-time expressions, then the *sn1*+*sn2*+ . . . can exceed *dn*, but any character code that would be stored in a character position beyond the end of the destination is discarded.

The destination of a CH$MOVE function must not overlap the source; that is, the two sequences must not have any character positions in common. Similarly, the destination of the CH$COPY function must not overlap any of its sources.

20.2.5.2. Examples

The sequence-writing functions are a convenience because they combine in a single function what would require many CH$WCHAR functions. Also, they contribute to efficiency by making use of the special hardware instructions especially designed for moving character sequences.

An example of the use of the CH$MOVE and CH$FILL functions follows:

```
OWN
    X: CH$SEQUENCE[20],
    P;
BIND
    S = UPLIT('ABCD');
...
P = CH$PTR(X);
INCR I FROM 1 TO 4 DO
    BEGIN
    P = CH$MOVE(.I, CH$PTR(S), .P);
    P = CH$FILL(%C'-', 5-.I, .P);
    END;
```

At the end of this fragment, the contents of X is as follows:

```
'A----AB---ABC--ABCD-'
```

The final value of P is a pointer to the twenty-first character position of X; that is, the unspecified character position that follows the last character position of X.

An example of the use of the CH$COPY function follows:

```
OWN
    ALPHA: CH$SEQUENCE[10];
BIND
    Q = UPLIT('ABCDEFGH');
...
CH$COPY(
    3, CH$PTR(Q,5),
    5, CH$PTR(Q),
    %C' ',
```

```
    10, CH$PTR(ALPHA));
```

At the end of this program fragment, the contents of ALPHA is as follows:

```
'FGHABCDE '
```

This example assigns a relatively complicated value to ALPHA by means of a single function call.

The CH$COPY function does not do anything that cannot be done by a combination of the CH $MOVE and CH$FILL functions. For example, the previous program fragment could be replaced by the following:

```
OWN
    ALPHA: CH$SEQUENCE[10],
    PA;
BIND
    Q = UPLIT('ABCDEFGH');
...
PA = CH$PTR(ALPHA);
PA = CH$MOVE(3, CH$PTR(Q,5), .PA);
PA = CH$MOVE(5, CH$PTR(Q), .PA);
CH$FILL(%C' ', 2, .PA);
```

This version is less compact and less efficient than the version that uses CH$COPY. The use of PA as temporary storage for the pointer could be eliminated by a nesting of function calls; nevertheless, this version would require three function calls to replace the single call on CH$COPY.

20.2.6. Sequence-Comparing Functions

Each of the sequence-comparing functions compares the contents of one character position sequence to another. With the exception of CH$COMPARE, these functions return 1 if the comparison is satisfied and return 0 otherwise; thus they serve character sequences in the same way relational operators serve integer and address values (see *Section 5.1.4.5, "Relational Expressions"*). If one of the character sequences is shorter than the other, it is extended (for purposes of the comparison only) by the addition of fill characters at the end.

The CH$EQL function determines whether or not the two given sequences are identical, and the CH $NEQ function is the negation of CH$EQL. The remaining- sequence comparing functions depend on the ordering of character sequences. That ordering is determined by rules similar to those for arranging the words and phrases in a dictionary. The CH$LSS function determines whether or not the first parameter occurs before the second parameter in the ordering of sequences. The CH$LEQ, CH$GTR, and CH$GEQ functions are similarly defined.

The CH$COMPARE function determines whether the first parameter occurs before, is equal to, or occurs after the second parameter. The function returns –1, 0, or 1, respectively. This function can be used as a case-index in a case-expression to provide, in a clear and efficient way, an action for each of the three possible relations between two sequences.

20.2.6.1. Definition

The sequence-comparing functions are defined as follows:

CH$*xxx*(*n1*, *ptr1*, *n2*, *ptr2*, *fill*)

In this definition, "CH$*xxx*" stands for any one of the seven function names given in the table below. Interpret *n1* and *n2* as unsigned integers (the lengths of the given sequences). Interpret *ptr1* and *ptr2*

as character pointers. Use these pointers to locate the beginning positions of two character position sequences. Interpret *fill* as a character code.

If *n1* is not equal to *n2* (so that the sequences are of different lengths), treat the shorter one as if it had sufficient additional character positions and each additional character position contained *fill*.

Look through the two sequences in parallel, one character position at a time. That is, select the first position of each sequence, then select the second position of each sequence, and so on. Proceed in this manner until a position is selected that contains one character code for one sequence and a different character code for the other. If no such position is found (because the sequences are identical), proceed to the last position of the sequences.

Call the character codes in the selected positions of the first and second sequence *c1* and *c2*, respectively. These character codes are integers, and are subject to arithmetic comparison. On the basis of the function name and the character codes *c1* and *c2*, obtain a value from the following table:

Function Name	c1 less than c2	c1 equal to c2	c1 greater than c2
CH$EQL	0	1	0
CH$NEQ	1	0	1
CH$LSS	1	0	0
CH$LEQ	1	1	0
CH$GTR	0	0	1
CH$GEQ	0	1	1
CH$COMPARE	-1	0	1

Return the value thus obtained.

Default fill character: The last parameter can be omitted; that is, the form CH$(*n1*,*ptr1*,*n2*,*ptr2*) is permitted. In this case, zero is used as the value of *fill*.

20.2.6.2. Examples

Assume the following declarations in the examples:

```
BIND
    P_ALPHA = CH$PTR(UPLIT('ALPHA')),
    P_BETA  = CH$PTR(UPLIT('BETA')),
    P_BEAR  = CH$PTR(UPLIT('BEAR')),
    P_BE    = CH$PTR(UPLIT('BE'));
```

The examples are as follows:

1. CH$LSS(5, P_ALPHA, 4, P_BETA)

   When corresponding characters are compared, it is determined that the first characters of the parameters, 'A' and 'B', are different. Because the ASCII code for 'A' is less than the ASCII code for 'B', the value of the function is 1.

2. CH$GTR(4, P_BETA, 4, P_BEAR)

It is determined that the third characters of the parameters 'T' and 'A' are different. Because the ASCII code for 'T' comes after the ASCII code for 'A', the value of the function is 1.

3. CH$GTR(4, P_BEAR, 2, P_BE)

   The fill character added to the second parameter plays a decisive role. That is, the first two characters of the parameters are the same, so it is 'A' and the fill character that are different. The default fill character is 0. Because the ASCII code for 'A' is greater than 0, the value of the function is 1.

4. CH$GTR(4, P_BEAR, 2, P_BE, 127)

   The fill character is given explicitly as 127, which is equal to the highest ASCII code. Because the ASCII code for 'A' is less than 127, the value of the function is 0.

5. CH$COMPARE(5, P_ALPHA, 4, P_BETA)

   Because the value of the ASCII code for 'A' is less than the ASCII code for 'B', the value of the function is –1.

20.2.7. Sequence-Searching Functions

The sequence-searching functions are used to find a single character or a sequence of characters within a larger character sequence. Searching is always done from left to right (from the first character position to the last).

The CH$FIND_CH function looks for a character position that contains a given character, whereas the CH$FIND_NOT_CH looks for a character position that contains anything but a given character. The CH $FIND_SUB function looks for a given sequence of characters.

If the desired character or character sequence cannot be found by these functions, a *null pointer* is returned. A CH$FAIL function then determines whether the returned pointer is or is not a null pointer. A null pointer must not be passed to any CH$ function except CH$FAIL.

20.2.7.1. Definition

The sequence-searching functions are defined as follows:

CH$FIND_CH(*n*, *ptr*, *char*)

Interpret *n* as an unsigned integer (the length of the context). Interpret *ptr* as a character pointer. Interpret *char* as a character code. Use *ptr* to locate a character sequence, the context.

Search the first *n* character positions of the context for a position that contains *char*, and return a pointer to that position. If no such character position is found, return the null pointer.

CH$FIND_NOT_CH(*n*, *ptr*, *char*)

Proceed as for CH$FIND_CH above. However, search the given sequence for a position whose contents are not equal to char.

CH$FIND_SUB(*cn*, *cptr*, *pn*, *pptr*)

Interpret *cn* and *pn* as unsigned integers (the lengths of the context and pattern, respectively). Interpret *cptr* and *pptr* as character pointers. Use these pointers to locate two character position sequences, the context and the pattern.

Start at the first character position of the context and search for a sequence of positions that contains the pattern. If such a sequence is found, return a character pointer to the first position of the sequence. Otherwise, return the null pointer.

CH$FAIL(*ptr*)

Interpret *ptr* as a pointer. If the pointer is the null pointer, then return 1; otherwise, return 0.

20.2.7.2. Examples

As an example of the use of the CH$FIND_CHAR and CH$FIND_NOT_CHAR functions, consider the following routine:

```
ROUTINE FIND_WORD(N, LINE): NOVALUE =
    BEGIN
    EXTERNAL ROUTINE
                    PROCESS_WORD;
OWN
                    LE,
                    RE;
LE = CH$FIND_NOT_CH(.N, .LINE, %C' ');
RE = CH$FIND_CH(.N-CH$DIFF(.LE,.LINE), .LE, %C' ');
PROCESS_WORD(CH$DIFF(.RE,.LE), .LE);
END;
```

This routine finds the first full "word" in a given line of text. For purposes of this routine, a "word" is any sequence of characters that does not contain a space.

The two parameters of the routine are defined as follows:

.N	The number of positions in the character position sequence that contains the given text
.LINE	A pointer to the first position of the character position sequence that contains the given text

The first assignment in the routine sets .LE to point to the first character of the word. The second assignment sets .RE to point to the first space after the word. Finally, a routine that processes the word is called; that routine, PROCESS_ WORD, is not specified here.

20.2.8. Sequence-Translating Functions

The sequence-translating functions are used to translate a character sequence from one encoding to another. The CH$TRANSTABLE function builds a table that controls the translation. The CH $TRANSLATE function uses the table to translate a given sequence into the new encoding.

The *character translation table* is, itself, a character position sequence. Suppose, for example, the contents of the first character position of such a table is 7; this means that a character code whose value is 0 will be translated to 7 by the table.

The table contains one position for each character-code value in the source character-code set. For example, if the source character sequence is ASCII encoded, then the translation table must contain 128 positions, one for each value in the (7-bit) ASCII character-code set. The CH$TRANSLATE function uses the value of a given source character position as a zero-based index into the table, from which it obtains the corresponding destination code value.

20.2.8.1. Definition

The contents of a character translation table is given as a parameter of the CH$TRANSTABLE function. The syntax of this parameter is:

```
translation-string      translation-item , . . .

translation-item        { translation-code                          }
                        { REP replicator OF ( translation-string ) }

replicator              compile-time-constant-expression

translation-code        single-character-literal
```

The sequence-translating functions are defined as follows:

CH$TRANSTABLE(*ts*)

The symbol *ts* represents a *translation-string*, which is described above. Create the translation table specified by *ts* and place it in the current PLIT program section. Return the address of the translation table.

CH$TRANSLATE(*tab*, *sn*, *sptr*, *fill*, *dn*, *dptr*)

Interpret *tab* as an address and use it to locate a character translation table. Interpret *sn* and *dn* as unsigned integers (the lengths of the source and the destination, respectively). Interpret *sptr* and *dptr* as pointers and use them to locate the beginning positions of two character position sequences (the source and the destination). Interpret *fill* as a character code.

Let *n* be *sn* or *dn* (the length of the source or the destination), whichever is smaller. Perform the following steps for $i = 1, 2, \ldots, n$: fetch the contents of the *i*th character position of the source, and call its value *c*. Fetch the contents of character position *c* of the character translation table (whose first position is numbered zero), and call the value *tc*. Store *tc* in the *i*th character position of the destination.

If *sn* is greater than *dn* (that is, the source is longer than the destination), then ignore the last *sn*-*dn* positions of the source. If *sn* is less than *dn*, then set the last (*dn*-*sn*) character positions of the destination to *fill*. Observe that the *fill* character code is not translated.

Return a pointer to the (*dn*+1)th character position of the destination.

The CH$TRANSTABLE function is always a compile-time constant expression. In fact, the table is created and allocated by the compiler in the same way a PLIT is created and allocated. The destination of a call on the CH$TRANSLATE function must not overlap the source; that is, the two sequences must not have any character positions in common.

20.2.8.2. Examples

As an example of the use of the sequence-translating functions, consider the following routine:

```
ROUTINE R(N, LINE, WORK_BUF): NOVALUE =
    BEGIN
```

```
    BIND
              TAB =
                  CH$TRANSTABLE(
                   REP 32 OF (%C'*'),
                   %C' ',
                   REP 10 OF (%C'*'),
                   %C'+',
                   REP 1 OF (%C'*'),
                   %C'-',
                   REP 2 OF (%C'*'),
                   %C'0', %C'1', %C'2', %C'3', %C'4',
                   %C'5', %C'6', %C'7', %C'8', %C'9',
                   REP 70 OF (%C'*'));
...
CH$TRANSLATE(
              TAB,
              .N, .LINE,
              0,
              .N, .WORK_BUF);
STAR = CH$FIND_CH(.N, .WORK_BUF, %C'*');
IF CH$FAIL(.STAR)
    THEN PROCESS(.N, .LINE)
    ELSE ERROR(.N, .LINE, CH$DIFF(.STAR, .WORK_BUF));
END;
```

This routine performs a preliminary check of a given line of text that is expected to represent one or more integers. For purposes of this routine, the presence of any character other than a space, a sign, or a digit makes the line invalid. If the line is valid, then a routine to further process the line if called; that routine, PROCESS, is not specified here. Otherwise, a routine to handle an invalid line, ERROR, also not specified here, is called.

The three parameters of the routine are defined as follows:

.N	The number of positions in the character position sequence that contains the given text
.LINE	A pointer to the first position of the character position sequence that contains the given text
.WORK_BUF	A pointer to the first character position of a work area that is to receive the translated sequence

A step-by-step description of the routine R follows:

1. A translation table is defined and its address is bound to TAB. The table is designed to leave unchanged any space, sign, or decimal digit, but to replace any other character with an asterisk (*).
2. The given character position sequence is translated. If it is valid, it is unchanged. If it is invalid, each invalid character is replaced by an asterisk.
3. The translated sequence is searched for an asterisk, and the resulting pointer is assigned to STAR.
4. The pointer in STAR is checked by means of CH$FAIL. If it is null, then no asterisk was found and the text is passed to the routine PROCESS. If the pointer is not null, the line is passed to the error routine together with the index of the first invalid character.

This program fragment is relatively complicated, but it is very efficient. Without the translating functions, some method of checking individually for each of the valid characters would be required.

Appendix A. Predefined Identifiers

A **predefined identifier** is an identifier that has a special meaning in one or more dialects of BLISS. For example, "IF" indicates the beginning of a conditional-expression, and "MAXU" designates the "unsigned maximum" standard-function.

There are four kinds of predefined identifiers that are classified as either **keywords** or **predefined names**. Each keyword is either **reserved** or **unreserved**, and each predefined name is either **predeclared** or **built-in**.

The use of a predefined identifier as an explicitly declared name is more or less restricted, depending on the classification of the identifier. The restrictions are as follows:

- A **reserved keyword** must not be used as an explicitly declared name under any circumstances.
- An **unreserved keyword** can be used freely as an explicitly declared name, just as if it were not a predefined identifier. The only disadvantage is that a reader of a program may be confused to see a familiar BLISS keyword (such as MAIN, for example) being used as an explicitly declared name.
- A **predeclared name** can be used as an explicitly declared name. However, such a use makes it impossible to use the name in its predefined sense within the scope of the explicit declaration. For example, wherever ABS is explicitly declared (for example, as a data segment name), it cannot be used as the name of the absolute value standard-function.
- A **built-in name** must always appear in an explicit declaration. If it is declared by a built-in declaration, then it has its predefined meaning; otherwise, it has the meaning given it by the explicit declaration, just as if it were not a predefined identifier.

These restrictions can be summarized as follows: In choosing a name, never use a reserved keyword and avoid the use of any predefined name if its use could cause confusion.

The following list includes identifiers that are predefined in the versions of BLISS described in this manual, as well as a number of identifiers that will be predefined in later versions of BLISS. The applicable dialects are indicated by parenthesized numbers in the classification column.

Identifier	Classification	Usage
ABS	predeclared name	standard-function
ABSOLUTE	unreserved keyword(16,32)	addr.-mode, object-option
ACTUALCOUNT	built-in name	linkage-function
ACTUALPARAMETER	built-in name	linkage-function
ADDRESSING_MODE	reserved keyword	addr.-mode-attr., -switch
ALIGN	reserved keyword	alignment-attribute
ALWAYS	reserved keyword	select-label
AND	reserved keyword	operator-expression
AP	built-in name(32,36)	register-name
ARGPTR	built-in name	linkage-function
ASSEMBLY	unreserved keyword	list-option

BEGIN	reserved keyword	block
BINARY	unreserved keyword	list-option
BIND	reserved keyword	bind-declaration
BIT	reserved keyword	(Future BLISS)
BITVECTOR	predeclared name	structure-name
BLISS	predeclared name	linkage-name
BLISS10	predeclared name(36)	environment-option
BLISS10_OTS	unreserved keyword(36)	environment-option
BLISS16	unreserved keyword	language-name
BLISS32	unreserved keyword	language-name
BLISS36	unreserved keyword	language-name
BLISS36C	predeclared name(36)	linkage-name
BLISS36C_OTS	unreserved keyword(36)	environment-option
BLOCK	predeclared name	structure-name
BLOCKVECTOR	predeclared name	structure-name
BUILTIN	reserved keyword	built-in-declaration
BY	reserved keyword	indexed-loop
BYTE	reserved keyword	allocation-unit
CALL	unreserved keyword(16,32)	linkage-type
CASE	reserved keyword	case-expression
CH$A_RCHAR	predeclared name	supplementary-function
CH$A_WCHAR	predeclared name	supplementary-function
CH$ALLOCATION	predeclared name	supplementary-function
CH$COMPARE	predeclared name	supplementary-function
CH$COPY	predeclared name	supplementary-function
CH$DIFF	predeclared name	supplementary-function
CH$EQL	predeclared name	supplementary-function
CH$FAIL	predeclared name	supplementary-function
CH$FILL	predeclared name	supplementary-function
CH$FIND_CH	predeclared name	supplementary-function
CH$FIND_NOT_CH	predeclared name	supplementary-function
CH$FIND_SUB	predeclared name	supplementary-function
CH$GEQ	predeclared name	supplementary-function
CH$GTR	predeclared name	supplementary-function

CH$LEQ	predeclared name	supplementary-function
CH$LSS	predeclared name	supplementary-function
CH$MOVE	predeclared name	supplementary-function
CH$NEQ	predeclared name	supplementary-function
CH$PLUS	predeclared name	supplementary-function
CH$PTR	predeclared name	supplementary-function
CH$RCHAR	predeclared name	supplementary-function
CH$RCHAR_A	predeclared name	supplementary-function
CH$SIZE	predeclared name	supplementary-function
CH$TRANSLATE	predeclared name	supplementary-function
CH$TRANSTABLE	predeclared name	supplementary-function
CH$WCHAR	predeclared name	supplementary-function
CH$WCHAR_A	predeclared name	supplementary-function
CLEARSTACK	unreserved keyword(16,36)	linkage-option
CODE	unreserved keyword	module-switch
CODECOMMENT	reserved keyword	codecomment block
COMMENTARY	unreserved keyword	list-option
COMPILETIME	reserved keyword	compile-time-declaration
CONCATENATE	unreserved keyword	psect-attribute
DEBUG	unreserved keyword	module-switch
DECR	reserved keyword	indexed-loop
DECRA	reserved keyword	indexed-loop
DECRU	reserved keyword	indexed-loop
DO	reserved keyword	loop-expression
ELSE	reserved keyword	conditional-expression
ELUDOM	reserved keyword	module
EMT	unreserved keyword(16)	linkage-option
ENABLE	reserved keyword	enable-declaration
END	reserved keyword	block
ENTRY	unreserved keyword(36)	module-switch
ENVIRONMENT	unreserved keyword(36)	module-switch
EQL	reserved keyword	operator-expression
EQLA	reserved keyword	operator-expression

EQLU	reserved keyword	operator-expression
EQV	reserved keyword	operator-expression
ERRS	unreserved keyword	switch-item, module-switch
EXECUTE	unreserved keyword	psect-attribute
EXITLOOP	reserved keyword	exitloop-expression
EXPAND	unreserved keyword	list-option
EXTENDED	unreserved keyword(36)	environment-option
EXTERNAL	reserved keyword	address-mode-switch
FIELD	reserved keyword	field-declaration, -attribute
FORTRAN	predeclared name(16,32)	linkage-name
FORTRAN_FUNC	predeclared name	linkage-name
FORTRAN_SUB	predeclared name	linkage-name
FORWARD	reserved keyword	data-, routine-declaration
FP	built-in name(32,36)	register-name
FROM	reserved keyword	indexed-loop, case-expression
F10	predeclared name(36)	linkage-name
GENERAL	unreserved keyword(32)	addressing-mode
GEQ	reserved keyword	operator-expression
GEQA	reserved keyword	operator-expression
GEQU	reserved keyword	operator-expression
GLOBAL	reserved keyword	linkage-option, psect-attribute
GTR	reserved keyword	operator-expression
GTRA	reserved keyword	operator-expression
GTRU	reserved keyword	operator-expression
IDENT	unreserved keyword	module-switch
IF	reserved keyword	conditional-expression
INCR	reserved keyword	indexed-loop
INCRA	reserved keyword	indexed-loop
INCRU	reserved keyword	indexed-loop
INDIRECT	unreserved keyword(36)	addressing-mode
INITIAL	reserved keyword	initial-attribute
INRANGE	reserved keyword	case-label

INTERRUPT	unreserved keyword(16,32)	linkage-type
IOPAGE	reserved keyword	(Future BLISS)
IOT	unreserved keyword(16)	linkage-type
JSB	unreserved keyword(32)	linkage-type
JSR	unreserved keyword(16)	linkage-type
JSYS	unreserved keyword(36)	linkage-type
KEYWORDMACRO	reserved keyword	keyword-macro-declaration
KA10	unreserved keyword(36)	environment-option
KC10	unreserved keyword(36)	environment-option
KI10	unreserved keyword(36)	environment-option
KL10	unreserved keyword(36)	environment-option
KS10	unreserved keyword(36)	environment-option
LABEL	reserved keyword	label-declaration
LANGUAGE	unreserved keyword	switch-item, module-switch
LEAVE	reserved keyword	leave-expression
LEQ	reserved keyword	operator-expression
LEQA	reserved keyword	operator-expression
LEQU	reserved keyword	operator-expression
LIBRARY	reserved keyword	list-option, library-declaration
LINKAGE	reserved keyword	switch, linkage-declaration
LINKAGE_REGS	unreserved keyword(36)	linkage-option
LIST	unreserved keyword	switch-item, module-switch
LITERAL	reserved keyword	literal-declaration
LOCAL	reserved keyword	local-declaration, psect-attribute
LONG	reserved keyword	allocation-unit
LONG_RELATIVE	unreserved keyword(32)	addressing-mode
LSI11	unreserved keyword(16)	environment-option
LSS	reserved keyword	operator-expression
LSSA	reserved keyword	operator-expression
LSSU	reserved keyword	operator-expression
MACRO	reserved keyword	macro-declaration

MAIN	unreserved keyword	module-switch
MAP	reserved keyword	map-declaration
MAX	predeclared name	standard-function
MAXA	predeclared name	standard-function
MAXU	predeclared name	standard-function
MIN	predeclared name	standard-function
MINA	predeclared name	standard-function
MINU	predeclared name	standard-function
MOD	reserved keyword	operator-expression
MODULE	reserved keyword	module
NEQ	reserved keyword	operator-expression
NEQA	reserved keyword	operator-expression
NEQU	reserved keyword	operator-expression
NOASSEMBLY	unreserved keyword	list-option
NOBINARY	unreserved keyword	list-option
NOCODE	unreserved keyword	module-switch
NOCOMMENTARY	unreserved keyword	list-option
NODEBUG	unreserved keyword	module-switch
NODEFAULT	unreserved keyword	psect-attribute
NOERRS	unreserved keyword	switch-item, module-switch
NOEXECUTE	unreserved keyword	psect-attribute
NOEXPAND	unreserved keyword	list-option
NOINDIRECT	unreserved keyword(36)	addressing-mode
NOLIBRARY	unreserved keyword	list-option
NONEXTERNAL	unreserved keyword(32)	addressing-mode-switch
NOOBJECT	unreserved keyword	list-option
NOOPTIMIZE	unreserved keyword	switch-item, module-switch
NOPIC	unreserved keyword	psect-attribute
NOPRESERVE	unreserved keyword	linkage-option
NOREAD	unreserved keyword	psect-attribute
NOREQUIRE	unreserved keyword	list-option
NOSAFE	unreserved keyword	switch-item, module-switch
NOSHARE	unreserved keyword	psect-attribute
NOSOURCE	unreserved keyword	list-option

NOSYMBOLIC	unreserved keyword	list-option
NOT	reserved keyword	operator-expression
NOTRACE	unreserved keyword	list-option
NOTUSED	unreserved keyword(32)	linkage-option
NOUNAMES	unreserved keyword	switch-item, module-switch
NOVALUE	reserved keyword	novalue-attribute
NOWRITE	unreserved keyword	psect-attribute
NOZIP	unreserved keyword	switch-item, module-switch
NULLPARAMETER	built-in name(16,32)	linkage-function
OBJECT	unreserved keyword	list-option, module-switch
OF	reserved keyword	case-, select-expression; plit
OPTIMIZE	unreserved keyword	switch-item, module-switch
OPTLEVEL	unreserved keyword	module-switch
OR	reserved keyword	operator-expression
ORIGIN	unreserved keyword(36)	psect-attribute
OTHERWISE	reserved keyword	select-label
OTS	unreserved keyword(36)	module-switch
OTS_LINKAGE	unreserved keyword(36)	module-switch
OUTRANGE	reserved keyword	case-label
OVERLAY	unreserved keyword	psect-attribute
OWN	reserved keyword	own-declaration
PC	built-in name(16,32)	register-name
PIC	unreserved keyword	psect-attribute
PLIT	reserved keyword	plit
PORTAL	unreserved keyword(36)	linkage-option
PRESERVE	unreserved keyword	linkage-option
PRESET	reserved keyword	preset-attribute
PSECT	reserved keyword	psect-declaration, -allocation
PS_INTERRUPT	unreserved keyword(36)	linkage-type
PUSHJ	unreserved keyword(36)	linkage-type
READ	unreserved keyword	psect-attribute
RECORD	reserved keyword	(Future BLISS)

REF	reserved keyword	structure-attribute
REGISTER	reserved keyword	register-, linkage-declaration
RELATIVE	unreserved keyword(16)	addressing-mode
RELOCATABLE	unreserved keyword(16)	object-option
REP	reserved keyword	plit
REQUIRE	reserved keyword	list-option, require-declaration
RETURN	reserved keyword	return-expression
ROUTINE	reserved keyword	routine-declaration
RSX_AST	unreserved keyword(16)	linkage-type
RTT	unreserved keyword(16)	linkage-option
R0	built-in name(16,32)	register-name
R1	built-in name(16,32)	register-name
R2	built-in name(16,32)	register-name
R3	built-in name(16,32)	register-name
R4	built-in name(16,32)	register-name
R5	built-in name(16,32)	register-name
R6	built-in name(32)	register-name
R7	built-in name(32)	register-name
R8	built-in name(32)	register-name
R9	built-in name(32)	register-name
R10	built-in name(32)	register-name
R11	built-in name(32)	register-name
SAFE	unreserved keyword	switch-item, module-switch
SELECT	reserved keyword	select-expression
SELECTA	reserved keyword	select-expression
SELECTONE	reserved keyword	select-expression
SELECTONEA	reserved keyword	select-expression
SELECTONEU	reserved keyword	select-expression
SELECTU	reserved keyword	select-expression
SET	reserved keyword	case-, select-expression; field-declaration
SETUNWIND	predeclared name	condition-handling-function
SHARE	unreserved keyword	psect-attribute
SHOW	reserved keyword	(Future BLISS)

SIGN	predeclared name	standard-function
SIGNAL	predeclared name	condition-handling-function
SIGNAL_STOP	predeclared name	condition-handling-function
SIGNED	reserved keyword	extension-, range-attribute
SKIP	unreserved keyword(36)	linkage-option
SOURCE	unreserved keyword	list-option
SP	built-in name	register-name
STACK	unreserved keyword(36)	environment-option
STACKLOCAL	reserved keyword	stacklocal-declaration
STANDARD	unreserved keyword	linkage-declaration
STANDARD_OTS	unreserved keyword(36)	environment-option
STRUCTURE	reserved keyword	structure-declaration, switch
SWITCHES	reserved keyword	switches-declaration
SYMBOLIC	unreserved keyword	list-option
T11	unreserved-keyword(16)	environment-option
TES	reserved keyword	case-, select-expression; field-declaration
THEN	reserved keyword	conditional-expression
TO	reserved keyword	loop, case-expression, select-label
TOPS10	unreserved keyword(36)	environment-option
TOPS20	unreserved keyword(36)	environment-option
TRACE	unreserved keyword	list-option
TRAP	unreserved keyword(16)	linkage-type
UNAMES	unreserved keyword	switch-item, module-switch
UNDECLARE	reserved keyword	undeclare-declaration
UNSIGNED	reserved keyword	extension-, range-attribute
UNTIL	reserved keyword	loop-expression
UPLIT	reserved keyword	plit
VALUECBIT	unreserved keyword(16)	linkage-option
VECTOR	predeclared name	structure-name, psect-attr.
VERSION	unreserved keyword	switch-item, module-switch
VOLATILE	reserved keyword	volatile-attribute

XOR	reserved keyword	operator-expression
ZIP	unreserved keyword	switch-item, module-switch
$CODE$	predeclared name	psect-name
$GLOBAL$	predeclared name	psect-name
$HIGH$	predeclared name(36)	psect-name
LOW	predeclared name(36)	psect-name
OWN	predeclared name	psect-name
$PLIT$	predeclared name	psect-name
%ALLOCATION	reserved keyword	allocation-function
%ASCIC	reserved keyword(16,32)	string-literal
%ASCID	reserved keyword	string-literal
%ASCII	reserved keyword	string-literal
%ASCIZ	reserved keyword	string-literal
%ASSIGN	reserved keyword	calculation-function
%B	reserved keyword	integer-literal
%BLISS	reserved keyword	compiler-state-function
%BLISS16	reserved keyword	predeclared macro
%BLISS32	reserved keyword	predeclared macro
%BLISS36	reserved keyword	predeclared macro
%BPADDR	reserved keyword	predeclared literal
%BPUNIT	reserved keyword	predeclared literal
%BPVAL	reserved keyword	predeclared literal
%C	reserved keyword	integer-literal
%CHAR	reserved keyword	string-function
%CHARCOUNT	reserved keyword	string-function
%COUNT	reserved keyword	macro-function
%CTCE	reserved keyword	exp-test-function
%D	reserved keyword	float-literal

%DECIMAL	reserved keyword	integer-literal
%DECLARED	reserved keyword	compiler-state-function
%E	reserved keyword	float-literal
%ELSE	reserved keyword	lexical-conditional
%ERROR	reserved keyword	advisory-function
%ERRORMACRO	reserved keyword	advisory-function
%EXACTSTRING	reserved keyword	string-function
%EXITITERATION	reserved keyword	macro-function
%EXITMACRO	reserved keyword	macro-function
%EXPAND	reserved keyword	quote-function
%EXPLODE	reserved keyword	delimiter-function
%FI	reserved keyword	lexical-conditional
%FIELDEXPAND	reserved keyword	fieldexpand-function
%G	reserved keyword	float-literal
%H	reserved keyword	float-literal
%IDENTICAL	reserved keyword	sequence-test-function
%IF	reserved keyword	lexical-conditional
%INFORM	reserved keyword	advisory-function
%ISSTRING	reserved keyword	exp-test-function
%LENGTH	reserved keyword	macro-function
%LTCE	reserved keyword	exp-test-function
%MESSAGE	reserved keyword	advisory-function
%NAME	reserved keyword	name-function
%NBITS	reserved keyword	bits-function
%NBITSU	reserved keyword	bits-function
%NULL	reserved keyword	sequence-test-function
%NUMBER	reserved keyword	calculation-function

%O	reserved keyword	integer-literal
%P	reserved keyword	string-literal
%PRINT	reserved keyword	advisory-function
%QUOTE	reserved keyword	quote-function
%QUOTENAME	reserved keyword	macro-name function
%RAD50_10	reserved keyword(36)	string-literal
%RAD50_11	reserved keyword(16,32)	string-literal
%REF	reserved keyword	standard-function
%REMAINING	reserved keyword	macro-function
%REMOVE	reserved keyword	delimiter-function
%REQUIRE	reserved keyword	require-function
%SBTTL	reserved keyword	title-function
%SIXBIT	reserved keyword(36)	string-literal
%SIZE	reserved keyword	allocation-function
%STRING	reserved keyword	string-function
%SWITCHES	reserved keyword	compiler-state-function
%THEN	reserved keyword	lexical-conditional
%TITLE	reserved keyword	title-function
%UNQUOTE	reserved keyword	quote-function
%UPVAL	reserved keyword	predeclared literal
%VARIANT	reserved keyword	compiler-state-function
%WARN	reserved keyword	advisory-function
%X	reserved keyword	integer-literal

Appendix B. String Encodings

This appendix describes the several types of character-string encodings used in the BLISS dialects:

- In BLISS–16 and BLISS–32: ASCII and RAD50_11
- In BLISS–36: ASCII, RAD50_10, and SIXBIT

B.1. ASCII Encoding

An ASCII string-literal is a common way of encoding a character sequence. The size of an ASCII character position varies with the dialect as follows: In BLISS–16 and BLISS–32, one character occupies an 8-bit byte; in BLISS–36, each 36-bit word contains five ASCII character positions, each of which occupies seven bits.

The code value for each ASCII character can be found in B–1 both in octal and hexadecimal representation.

Table B.1. ASCII Code Table

Octal Code	Hex Code	ASCII Char.	Octal Code	Hex Code	ASCII Char.	Octal Code	Hex Code	ASCII Char.
000	00	NUL	053	2B	+	126	56	V
001	01	SOH	054	2C	,	127	57	W
002	02	STX	055	2D	–	130	58	X
003	03	ETX	056	2E	.	131	59	Y
004	04	EOT	057	2F	/	132	5A	Z
005	05	ENQ	060	30	0	133	5B	[
006	06	ACK	061	31	1	134	5C	\
007	07	BEL	062	32	2	135	5D	]
010	08	BS	063	33	3	136	5E	^
011	09	HT	064	34	4	137	5F	_
012	0A	LF	065	35	5	140	60	`
013	0B	VT	066	36	6	141	61	a
014	0C	FF	067	37	7	142	62	b
015	0D	CR	070	38	8	143	63	c
016	0E	SO	071	39	9	144	64	d
017	0F	SI	072	3A	:	145	65	e
020	10	DLE	073	3B	;	146	66	f
021	11	DC1	074	3C	<	147	67	g
022	12	DC2	075	3D	=	150	68	h
023	13	DC3	076	3E	>	151	69	i

Octal Code	Hex Code	ASCII Char.	Octal Code	Hex Code	ASCII Char.	Octal Code	Hex Code	ASCII Char.
024	14	DC4	077	3F	?	152	6A	j
025	15	NAK	100	40	@	153	6B	k
026	16	SYN	101	41	A	154	6C	l
027	17	ETB	102	42	B	155	6D	m
030	18	CAN	103	43	C	156	6E	n
031	19	EM	104	44	D	157	6F	o
032	1A	SUB	105	45	E	160	70	p
033	1B	ESC	106	46	F	161	71	q
034	1C	FS	107	47	G	162	72	r
035	1D	GS	110	48	H	163	73	s
036	1E	RS	111	49	I	164	74	t
037	1F	US	112	4A	J	165	75	u
040	20	space	113	4B	K	166	76	v
041	21	!	114	4C	L	167	77	w
042	22	"	115	4D	M	170	78	x
043	23	#	116	4E	N	171	79	y
044	24	$	117	4F	O	172	7A	z
045	25	%	120	50	P	173	7B	{
046	26	&	121	51	Q	174	7C	|
047	27	'	122	52	R	175	7D	}
050	28	(	123	53	S	176	7E	~
051	29	)	124	54	T	177	7F	DEL
052	2A	*	125	55	U			

B.2. Radix–50 Encoding

A Radix–50 string-literal specifies a particular way of encoding and packing a sequence of characters. The characters in the string-literal must be members of the Radix–50 character set, which is a 40-character subset of the ASCII graphic characters. This subset is the same for all three BLISS dialects, but the details of encoding and packing vary between BLISS–16 and BLISS–32 on one hand (RAD50_11) and BLISS–36 on the other (RAD50_10). These two variations of Radix–50 encoding are described in the following two subsections.

B.2.1. RAD50_11 Encoding

In BLISS–16 and BLISS–32, Radix–50 encoding is invoked with the %RAD50_11 string function (see *Section 4.3, "String Literals"*). A sequence of Radix–50 characters is packed three characters per 16-bit word, as described below.

If necessary, trailing blanks are added so that the number of characters in the sequence is a multiple of 3. Then the sequence is divided into groups of three characters. The code for each character is obtained from B–2, based on both the character and its position in its group. Then the octal codes for each character in a group are added together to obtain a 16-bit value.

For example, if the string-literal %RAD50_11'AB' is evaluated, a trailing blank is added, giving %RAD50_11'AB '. Then the literal is encoded and packed as follows:

```
A (as first character)          = 003100
B (as second character)         = 000120
Blank (as third character)      = 000000
            %RAD50_11'AB'       = 003220 (octal)
```

The character encoding table is derived as follows. The Radix–50 character set is composed of 50 (octal) characters. These characters are treated as the digits of a radix-50 number system. Suppose the ith character of the set must be encoded. Depending on whether it is the first (leftmost), second, or third character of a sequence, the character is encoded as 50*50*i, 50*i, or i (all octal). The value 50 (octal) was chosen as the radix because it is the largest value that permits the packing of three characters into a 16-bit word.

Table B.2. RAD50_11 Code Table

First Character		**Second Character**		**Third Character**	
Blank	000000	Blank	000000	Blank	000000
A	003100	A	000050	A	000001
B	006200	B	000120	B	000002
C	011300	C	000170	C	000003
D	014400	D	000240	D	000004
E	017500	E	000310	E	000005
F	022600	F	000360	F	000006
G	025700	G	000430	G	000007
H	031000	H	000500	H	000010
I	034100	I	000550	I	000011
J	037200	J	000620	J	000012
K	042300	K	000670	K	000013
L	045400	L	000740	L	000014
M	050500	M	001010	M	000015
N	053600	N	001060	N	000016
O	056700	O	001130	O	000017
P	062000	P	001200	P	000020
Q	065100	Q	001250	Q	000021
R	070200	R	001320	R	000022

First Character		Second Character		Third Character	
S	073300	S	001370	S	000023
T	076400	T	001440	T	000024
U	101500	U	001510	U	000025
V	104600	V	001560	V	000026
W	107700	W	001630	W	000027
X	113000	X	001700	X	000030
Y	116100	Y	001750	Y	000031
Z	121200	Z	002020	Z	000032
$	124300	$	002070	$	000033
.	127400	.	002140	.	000034
Unused	132500	Unused	002210	Unused	000035
0	135600	0	002260	0	000036
1	140700	1	002330	1	000037
2	144000	2	002400	2	000040
3	147100	3	002450	3	000041
4	152200	4	002520	4	000042
5	155300	5	002570	5	000043
6	160400	6	002640	6	000044
7	163500	7	002710	7	000045
8	166600	8	002760	8	000046
9	171700	9	003030	9	000047

B.2.2. RAD50_10 Encoding

In BLISS–36, Radix–50 encoding is invoked with the %RAD50_10 string function (see *Section 4.3, "String Literals"*). A sequence of Radix–50 characters is encoded and packed six characters per 36-bit word, as described below.

The sequence is divided into groups of six characters. If the last (or only) group contains less than six characters, leading blanks are added to the group in order to extend it to six characters. For each of these groups, the code for each character is obtained from B–3 and B–4, which list codes starting with the righthand character. (Note that this table has several differences from the RAD50_11 table.) Then these octal codes are added to obtain a 36-bit value.

For example, if the string-literal %RAD50_11 'ABCD' is evaluated, two leading blanks are added, giving %RAD50-11 ' ABCD'. Then the literal is encoded and packed as follows:

```
D (as rightmost character)          = 000000000016
C (as second character from right)  = 000000001010
```

```
B (as third character from right)        = 000000045400
A (as fourth character from right)       = 000002537000
Blank (as fifth character from right)    = 000000000000
Blank (as sixth character from           = 000000000000
right)
            %RAD50_10' ABCD'             = 000002605426 (octal)
```

The RAD50_10 character encoding table is derived as follows. The Radix–50 character set is composed of 50 (octal) characters. These characters are treated as the digits of a Radix–50 number system. If the ith character of the set which is located as the nth character from the right in a group must be encoded, it is represented as (50**(n–1))*i (where numbers are octal and ** denotes exponentiation). Thus if six characters are numbered from right to left in the following form:

C(6) C(5) C(4) C(3) C(2) C(1)

where C(*n*) is the octal code for the nth character, the RAD50_10 representation of the character string can be generated by the following:

```
((((((C(6)*50)+C(5))*50+C(4))*50+C(3))*50+C(2))*50+C(1)
```

where all numbers are octal.

Table B.3. RAD50_10 Code Table

Rightmost Character Code		Second Character from Right		Third Character from Right	
Blank	000000000000	Blank	000000000000	Blank	000000000000
0	000000000001	0	000000000050	0	000000003100
1	000000000002	1	000000000120	1	000000006200
2	000000000003	2	000000000170	2	000000011300
3	000000000004	3	000000000240	3	000000014400
4	000000000005	4	000000000310	4	000000017500
5	000000000006	5	000000000360	5	000000022600
6	000000000007	6	000000000430	6	000000025700
7	000000000010	7	000000000500	7	000000031000
8	000000000011	8	000000000550	8	000000034100
9	000000000012	9	000000000620	9	000000037200
A	000000000013	A	000000000670	A	000000042300
B	000000000014	B	000000000740	B	000000045400
C	000000000015	C	000000001010	C	000000050500
D	000000000016	D	000000001060	D	000000053600
E	000000000017	E	000000001130	E	000000056700
F	000000000020	F	000000001200	F	000000062000
G	000000000021	G	000000001250	G	000000065100

Rightmost Character Code		Second Character from Right		Third Character from Right	
H	000000000022	H	000000001320	H	000000070200
I	000000000023	I	000000001370	I	000000073300
J	000000000024	J	000000001440	J	000000076400
K	000000000025	K	000000001510	K	000000101500
L	000000000026	L	000000001560	L	000000104600
M	000000000027	M	000000001630	M	000000107700
N	000000000030	N	000000001700	N	000000113000
O	000000000031	O	000000001750	O	000000116100
P	000000000032	P	000000002020	P	000000121200
Q	000000000033	Q	000000002070	Q	000000124300
R	000000000034	R	000000002140	R	000000127400
S	000000000035	S	000000002210	S	000000132500
T	000000000036	T	000000002260	T	000000135600
U	000000000037	U	000000002330	U	000000140700
V	000000000040	V	000000002400	V	000000144000
W	000000000041	W	000000002450	W	000000147100
X	000000000042	X	000000002520	X	000000152200
Y	000000000043	Y	000000002570	Y	000000155300
Z	000000000044	Z	000000002640	Z	000000160400
.	000000000045	.	000000002710	.	000000163500
$	000000000046	$	000000002760	$	000000166600
%	000000000047	%	000000003030	%	000000171700

Table B.4. RAD50_10 Code Table

Fourth Character from Right		Fifth Character from Right		Sixth Character from Right	
Blank	000000000000	Blank	000000000000	Blank	000000000000
0	000000175000	0	000011610000	0	000606500000
1	000000372000	1	000023420000	1	001415200000
2	000000567000	2	000035230000	2	002223700000
3	000000764000	3	000047040000	3	003032400000
4	000001161000	4	000060650000	4	003641100000
5	000001356000	5	000072460000	5	004447600000
6	000001553000	6	000104270000	6	005256300000
7	000001750000	7	000116100000	7	006065000000

Fourth Character from Right		Fifth Character from Right		Sixth Character from Right	
8	000002145000	8	000127710000	8	006673500000
9	000002342000	9	000141520000	9	007502200000
A	000002537000	A	000153330000	A	010310700000
B	000002734000	B	000165140000	B	011117400000
C	000003131000	C	000176750000	C	011726100000
D	000003326000	D	000210560000	D	012534600000
E	000003523000	E	000222370000	E	013343300000
F	000003720000	F	000234200000	F	014152000000
G	000004115000	G	000246010000	G	014760500000
H	000004312000	H	000257620000	H	015567200000
I	000004507000	I	000271430000	I	016375700000
J	000004704000	J	000303240000	J	017204400000
K	000005101000	K	000315050000	K	020013100000
L	000005276000	L	000326660000	L	020621600000
M	000005473000	M	000340470000	M	021430300000
N	000005670000	N	000352300000	N	022237000000
O	000006065000	O	000364110000	O	023045500000
P	000006262000	P	000375720000	P	023654200000
Q	000006457000	Q	000407530000	Q	024462700000
R	000006654000	R	000421340000	R	025271400000
S	000007051000	S	000433150000	S	026100100000
T	000007246000	T	000444760000	T	026706600000
U	000007443000	U	000456570000	U	027515300000
V	000007640000	V	000470400000	V	030324000000
W	000010035000	W	000502210000	W	031132500000
X	000010232000	X	000514020000	X	031741200000
Y	000010427000	Y	000525630000	Y	032547700000
Z	000010624000	Z	000537440000	Z	033356400000
.	000011021000	.	000551250000	.	034165100000
$	000011216000	$	000563060000	$	034773600000
%	000011413000	%	000574670000	%	035602300000

B.3. Sixbit Encoding

In BLISS–36, SIXBIT encoding is invoked with the %SIXBIT string function (see *Section 4.3, "String Literals"*). SIXBIT encoding applies to the 64-character graphic subset of the ASCII characters. A sequence of SIXBIT characters is encoded as follows.

A character-sequence is divided into groups of six characters, with trailing blanks added to fill the final (or only) group of six, if necessary. Lowercase letters are converted to uppercase and then the 6-bit character code found in Table B-5 is obtained for each character. These six 6-bit codes form a fullword (36-bits).

Table B.5. SIXBIT Code Table

Octal Code	SIXBIT Char	Octal Code	SIXBIT Char	Octal Code	SIXBIT Char
00	space	25	5	53	K
01	!	26	6	54	L
02	"	27	7	55	M
03	#	30	8	56	N
04	$	31	9	57	O
05	%	32	:	60	P
06	&	33	;	61	Q
07	'	34	<	62	R
10	(	35	=	63	S
11	)	36	>	64	T
12	*	37	?	65	U
13	+	40	@	66	V
14	,	41	A	67	W
15	–	42	B	70	X
16	.	43	C	71	Y
17	/	44	D	72	Z
20	0	45	E	73	[
21	1	46	F	74	\
22	2	47	G	75	]
23	3	50	I	76	^
24	4	51	J	77	_
25	5	52	K		

Appendix C. Transportability Checking

This appendix describes the transportability checking that is performed by each compiler in response to the LANGUAGE special-switch. See *Section 18.2, "Switches-Declarations"* and *Section 19.2, "Module-Switches"* for the description of the LANGUAGE switch, and particularly *Section 18.2.5, "Discussion"* for a general discussion of its use.

When transportability checking is performed, the compiler scans the source input for any of the language features described below, and issues a warning message reporting any occurrence of such features. Two classes of transportability checking are currently provided, depending on how the language-list is specified in the LANGUAGE switch. The two classes are as follows:

1. Full Transportability Checking is performed if any one of the following specifications appears in the language-list:

   COMMON
   BLISS16,BLISS36
   BLISS32,BLISS36
   BLISS16,BLISS32,BLISS36

   All dialectal constructs are checked, as well as any other construct likely to cause problems in transporting a program between any two target systems.

2. BLISS–16/BLISS–32 Subset Checking is performed if the specification BLISS16, BLISS32 appears in the language list. This is a somewhat relaxed form of full (that is, Common BLISS) checking. Certain dialectal features that are valid in both BLISS–16 and BLISS–32 are not checked for in this case.

When no LANGUAGE switch appears in the module-head, or when a switch that specifies or implies only one language-name appears in either the module-head or a SWITCHES declaration, no transportability checking is done within the module or within the scope of the declaration, respectively (except that the switch specification, if explicit, is checked for validity). If specified, the LANGUAGE switch must include (or imply) the language-name corresponding to the compiler in use.

The specific language constructs involved in full checking and in BLISS–16/BLISS–32 subset checking are described in separate sections below.

C.1. Full Transportability Checking

The dialectal or problematic language features checked for and reported on under full checking are categorized below in alphabetical order.

Attributes: The dialectal attributes are as follows:

- Addressing-mode attribute
- Alignment-attribute
- Allocation-units BYTE, WORD, and LONG
- Extension-attributes SIGNED and UNSIGNED (when used as extension- attributes, see note below)

- Weak-attribute

Note

The keyword SIGNED or UNSIGNED when used as part of a range- attribute in a literal-declaration is a Common BLISS construct.

Built-in names and declarations: The occurrence, in a BUILTIN declaration, of any built-in-name except ACTUALCOUNT or ACTUALPARAMETER (common linkage-functions) is reported.

Condition handling features: Any use of an ENABLE declaration expression is reported. Use of parameters to SETUNWIND is reported.

Field selectors: Any field-selector that specifies a field not entirely contained within a fullword is reported. (That is, the position and size values must not exceed %BPVAL, and neither must their sum.) Also, any field-reference that does not modify a fetch or store operation and whose position value is not zero is reported. Note that the field-selector parameters must be compile-time constant expressions for the compiler to perform this checking.

GLOBAL and EXTERNAL names: The occurrence of any global- or external- name that is not unique (throughout the module) within its first six characters is reported.

Linkage declarations: Any use of a linkage-declaration is reported.

Linkage switches and linkage attributes: The use of any linkage-name other than FORTRAN_FUNC or FORTRAN_SUB in a linkage-switch or linkage-attribute is reported.

Literals: Occurrences of the following kinds of literals are reported:

- %E, %D, %G, and %H numeric-literals (floating point) and %P string-literals (packed decimal)
- Any string-literal used as a primary expression (that is, not a plit-item)
- An alphanumeric string-literal with a string-type other than %ASCII or %ASCIZ.

Module-switches: The occurrences of any of the following module-switches are reported:

PSECT declarations: Any use of a PSECT declaration is reported.

ADDRESSING_MODE OTS OTS_LINKAGE

Switches: The occurrence of ADDRESSING_MODE in a SWITCHES declaration is reported.

C.2. BLISS–16/BLISS–32 Subset Checking

The slightly less restrictive set of language features (relative to full checking) checked for and reported on under BLISS–16/BLISS–32 subset checking is categorized below in alphabetical order.

Briefly, the allocation-units BYTE and WORD, the extension-units SIGNED and UNSIGNED, and the string-type %RAD50_11 are considered transportable constructs in this case.

Attributes: The attributes checked on are as follows:

- Addressing-mode attribute

- Alignment-attribute
- Allocation-unit LONG
- Weak-attribute

Built-in names and declarations: The occurrence, in a BUILTIN declaration, of any built-in name except ACTUALCOUNT or ACTUALPARAMETER (common linkage-functions) is reported.

Condition-handling features: Any use of an ENABLE declaration is reported.

Field selectors: Any field-selector that specifies a field not entirely contained within a fullword is reported. That is, the position and size values must not exceed %BPVAL, and neither must their sum. Also, any field-reference that does not modify a fetch or store operation and whose position value is not zero is reported. Note that the field-selector parameters must be compile-time constant expressions for the compiler to perform this checking.

GLOBAL and EXTERNAL names: The occurrence of any global- or external- name that is not unique (throughout the module) within its first six characters is reported.

Linkage declarations: Any use of a linkage-declaration is reported.

Linkage switches and linkage attributes: The use of any linkage-name other than BLISS, FORTRAN, FORTRAN_FUNC, or FORTRAN_SUB in a linkage-switch or linkage- attribute is reported.

Literals: Occurrences of the following kinds of literals are reported:

- %E, %D, %G, and %H numeric-literals (floating point) and %P string-literals (packed decimal)
- Any string-literal used as a primary expression (that is, not a plit-item)
- An "alphanumeric" string-literal with a string-type other than %ASCII, %ASCIZ, or %RAD50_11.

Module-switches: The occurrences of any of the following module-switches are reported:

PSECT declarations: Any use of a PSECT declaration is reported. ADDRESSING_MODE OTS OTS_LINKAGE

Switches: The occurrence of ADDRESSING_MODE in a SWITCHES declaration is reported.

Appendix D. Built-In Functions

This appendix lists the names of the built-in machine-specific functions predefined for each BLISS dialect. Detailed descriptions of these functions may be found in the user manual associated with each BLISS dialect.

D.1. BLISS–16 Machine-Specific Functions

The following lists the predefined BLISS–32 machine-specific functions by operation.

D.1.1. Memory Management Operations

MFPD	Move from previous data space
MTPD	Move to previous data space
MFPI	Move from previous instruction space
MTPI	Move to previous instruction space

D.1.2. Processor Status Operations

MFPS	Move byte from processor status word
MTPS	Move byte to processor status word
SPL	Set priority level

D.1.3. Bit Manipulation Operations

ROT	Rotate
SWAB	Swap bytes

D.1.4. Arithmetic Operations

ADDD	Add D-floating operands
ADDF	Add F-floating operands
ADDM	Add multiword operands
DIVD	Divide D-floating operands
DIVF	Divide F-floating operands
EDIV	Extended-precision divide
EMUL	Extended-precision multiply
MULD	Multiply D-floating operands

MULF	Multiply F-floating operands
SUBD	Subtract D-floating operands
SUBF	Subtract F-floating operands
SUBM	Subtract multiword operands

D.1.5. Arithmetic Comparison Operations

CMPD	Compare D-floating operands
CMPF	Compare F-floating operands
CMPM	Compare multiword operands

D.1.6. Arithmetic Conversion Operations

CVTDF	Convert D-floating to F-floating
CVTFD	Convert F-floating to D-floating
CVTDI	Convert D-floating to integer
CVTID	Convert integer to D-floating
CVTFI	Convert F-floating to integer
CVTIF	Convert integer to F-floating

D.1.7. Processor Action Operations

BPT	Breakpoint trap
HALT	Halt processor
NOP	No operation
RESET	Reset hardware
WAIT	Processor wait

D.1.8. Miscellaneous Operations

DECX	Specialized routine call

D.2. BLISS–32 Machine-Specific Functions

The following lists the predefined BLISS–32 machine-specific functions by operation.

D.2.1. Processor Register Operations

MFPR	Move from a processor register
MTPR	Move to a processor register

D.2.2. Parameter Validation Operations

PROBER	Probe read accessibility
PROBEW	Probe write accessibility

D.2.3. Program Status Operations

BICPSW	Bit clear processor status word
BISPSW	Bit set processor status word
MOVPSL	Move from processor status longword

D.2.4. Queue Operations

INSQHI	Insert entry in queue head, interlocked
REMQHI	Remove entry from queue head, interlocked
INSQTI	Insert entry in queue tail, interlocked
REMQTI	Remove entry from queue tail, interlocked
INSQUE	Insert entry in queue
REMQUE	Remove entry from queue

D.2.5. Bit Manipulation Operations

FFC	Find first clear bit
FFS	Find first set bit
TESTBITCC	Test for bit clear, then clear bit
TESTBITCS	Test for bit clear, then set bit
TESTBITSC	Test for bit set, then clear bit
TESTBITSS	Test for bit set, then set bit
TESTBITCCI	Test for bit clear, then clear bit interlocked
TESTBITSSI	Test for bit set, then set bit interlocked

D.2.6. Arithmetic Operations

ADAWI	Add aligned word interlocked
ADDD	Add D-floating operands
ADDF	Add F-floating operands

ADDG	Add G-floating operands
ADDH	Add H-floating operands
ADDM	Add multiword operands
ASHQ	Arithmetic shift quad
DIVD	Divide D-floating operands
DIVF	Divide F-floating operands
DIVG	Divide G-floating operands
DIVH	Divide H-floating operands
EDIV	Extended-precision divide
EMUL	Extended-precision multiply
MULD	Multiply D-floating operands
MULF	Multiply F-floating operands
MULG	Multiply G-floating operands
MULH	Multiply H-floating operands
SUBD	Subtract D-floating operands
SUBF	Subtract F-floating operands
SUBG	Subtract G-floating operands
SUBH	Subtract H-floating operands
SUBM	Subtract multiword operands

D.2.7. Arithmetic Comparison Operations

CMPD	Compare D-floating operands
CMPF	Compare F-floating operands
CMPG	Compare G-floating operands
CMPH	Compare H-floating operands
CMPM	Compare multiword operands

D.2.8. Arithmetic Conversion Operations

CVTDF	Convert D-floating to F-floating
CVTFD	Convert F-floating to D-floating
CVTDI	Convert D-floating to integer
CVTID	Convert integer to D-floating

CVTDL	Convert D-floating to long
CVTLD	Convert long to D-floating
CVTFG	Convert F-floating to G-floating
CVTGF	Convert G-floating to F-floating
CVTFH	Convert F-floating to H-floating
CVTHF	Convert H-floating to F-floating
CVTFI	Convert F-floating to integer
CVTIF	Convert integer to F-floating
CVTFL	Convert F-floating to long
CVTLF	Convert long to F-floating
CVTLG	Convert long to G-floating
CVTGL	Convert G-floating to long
CVTLH	Convert long to H-floating
CVTHL	Convert H-floating to long
CVTRDH	Convert rounded D-floating to H-floating
CVTRDL	Convert rounded D-floating to long
CVTRFL	Convert rounded F-floating to long
CVTRGH	Convert rounded G-floating to H-floating
CVTRGL	Convert rounded G-floating to long
CVTRHL	Convert rounded H-floating to long

D.2.9. Character String Operations

CMPC3	Compare characters 3 operand
CMPC5	Compare characters 5 operand
CRC	Calculate cyclic redundancy check
LOCC	Locate character
SKPC	Skip character
MOVC3	Move character 3 operand
MOVC5	Move character 5 operand
MOVTC	Move translated characters

MOVTUC	Move translated until character
MATCHC	Match characters
SCANC	Scan characters
SPANC	Span characters

D.2.10. 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
CVTSP	Convert leading separate numeric to packed
CVTPT	Convert packed to trailing numeric
CVTTP	Convert trailing numeric to packed
EDITPC	Edit packed to character string
MOVP	Move packed

D.2.11. Processor Action Operations

BPT	Breakpoint
CHM(x)	Change mode
HALT	Halt processor
NOP	No operation

D.2.12. Miscellaneous Operations

BUGL	Bugcheck with long operand
BUGW	Bugcheck with word operand
CALLG	Call with general argument list
INDEX	Compute index
ROT	Rotate

XFC	Extended function call

D.3. BLISS–36 Machine-Specific Functions

The following lists the predefined BLISS–32 machine-specific functions by operation.

D.3.1. Logical Operations

ASH	Arithmetically shift a value
FIRSTONE	Find the leftmost nonzero list in a value
LSH	Logically shift a value
ROT	Rotate a value

D.3.2. Byte Manipulation Operations

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

D.3.3. Arithmetic Operations

ADDD	Add D-floating operands

ADDF	Add F-floating operands
ADDG	Add G-floating operands
DIVD	Divide D-floating operands
DIVF	Divide F-floating operands
DIVG	Divide G-floating operands
MULD	Multiply D-floating operands
MULF	Multiply F-floating operands
MULG	Multiply G-floating operands
SUBD	Subtract D-floating operands
SUBF	Subtract F-floating operands
SUBG	Subtract G-floating operands

D.3.4. Arithmetic Comparison Operations

CMPD	Compare D-floating operands
CMPF	Compare F-floating operands
CMPG	Compare G-floating operands

D.3.5. Arithmetic Conversion Operations

CVTDF	Convert D-floating to F-floating
CVTFD	Convert F-floating to D-floating
CVTDI	Convert D-floating to integer
CVTID	Convert integer to D-floating
CVTFI	Convert F-floating to integer
CVTIF	Convert integer to F-floating
CVTGF	Convert G-floating to F-floating
CVTFG	Convert F-floating to G-floating
CVTGI	Convert G-floating to integer
CVTIG	Convert integer to G-floating

D.3.6. Machine Code Insertion Operations

MACHOP	Execute a DECsystem–10/20 instruction
MACHSKIP	Execute a DECsystem–10/20 instruction and record any skip

D.3.7. System Interface Operations

JSYS	Perform a TOPS–20 monitor call
UUO	Perform a TOPS–10 monitor call