decsystem10

BLISS-10
Programmer's Reference Manual

digital

decsystem10

BLISS-10 PROGRAMMER'S REFERENCE MANUAL

DEC-10-LBRMA-A-D

This document reflects the software as of Version 4 of the BLISS-10 Compiler.

digital equipment corporation · maynard, massachusetts

First Printing, April 1973
Revised: February 1974

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

CONTENTS

FIGURES

TABLES

PREFACE

This manual is a reference manual. It is intended to contain the complete specification of the BLISS-10 language syntax and semantics as processed by version 4 of the BLISS-10 compiler.

Although the attempt has been made to clearly explain all language concepts and terms which are not part of the standard vocabulary, this document is not intended to be a tutorial manual or a guide to efficient BLISS-10 programming techniques. Although it is not required, familiarity with ALGOL block structure, storage allocation, and recursion will assist in the understanding of this document. In fact, we recommend that an individual who is attempting to learn the BLISS-10 language and who has no exposure to the concepts of BLOCK STRUCTURE and NAME SCOPE consult one of the many available ALGOL tutorial documents before reading this manual.

The material in this manual, including but not limited to, construction times and operating speeds is for information purposes only. All such material is subject to change without notice. Consequently DEC makes no claim and shall not be liable for its accuracy.

ACKNOWLEDGEMENT

The following people from Carnegie-Mellon University are acknowledged to have contributed substantially to the BLISS-10 Manual: J. Apperson, C. Geschke, C. Wienstock, D. Wile, W. Wulf, and J. Zarrella.

The cooperation of faculty, staff, and graduate students at Carnegie-Mellon University is possible in part because of a Department of Computer Science orientation that pursues research and advanced development efforts of relevance to the total community. Digital Equipment Corporation acknowledges the considerable assistance given by the members of the Department of Computer Science in bringing BLISS-10 into being.

BLISS-10 SUPPORT

The BLISS-10 compiler and related files are in software support category 4. This means that no formal support is provided under any circumstances by Digital Equipment Corporation for this software. Furthermore, Digital Equipment Corporation makes no claim that this software will ever be supported in the future. This material is provided for information purposes only.

Note also that Digital Equipment Corporation reserves the right to change the specifications of this language in any way including such ways as would invalidate currently legal source syntax.

Note also that it is possible for this version of the compiler to abort, loop, or otherwise run out of control and precautions should be taken to guard against such occurrences. We do not wish to imply in any way that this is production quality software.

Since this language or a variant may become supported in the future, we would appreciate receiving your comments and problems via the Software Performance Report mechanism. We cannot at this time, however, promise to reply to any such Software Performance Reports.

INTRODUCTION

BLISS-10 is a programming language for the DECsystem-10. It is specifically intended to be used for implementing "System Software". As such, it differs from other languages in several significant ways.

1. Higher level languages derive their suitability for a particular problem area - FORTRAN/ALGOL for mathematics, SNOBOL for strings, GPSS for simulations, etc., - because they provide to the user a vocabulary and set of conventions appropriate to that problem area. "Implementation Languages" can be viewed in a similar way - as application languages where the application is a particular brand of hardware. As such, an implementation language must reflect the capabilities and architecture of its machine and not block or frustrate the programmer's use of these capabilities.

2. Input/output is not a part of BLISS-10. I/O can be done directly either in the language much as an assembly program might or through subroutine calls.

3. Every attempt has been made to give the user explicit control over the code his program generates, while providing maximum convenience otherwise.

4. There are no explicit or implicit data modes other than the 36 bit binary word. Data modes are essentially user defined via the STRUCTURE(1) mechanism which allows the user to set or compute an algorithm for data access.

----

(1)Words in upper case represent reserved words in terminal symbols in the language. In programs the compiler recognizes them as reserved words in upper case only.

NOTE

BLISS-10 is in transition. This document reflects the target specification at the conclusion of the transition period. For those intending to use the language during the transition period, exception paragraphs are included immediately following the text which they modify. For example, if paragraph 1.5 contains the form of the final specification but it is presently not implemented, the paragraph

1.5 Exception

describes the current state of the language.

CHAPTER 1
LANGUAGE DEFINITION

1.1 AN EXPRESSION LANGUAGE

The programming language BLISS-10 enables programmers (persons who converse in a programming language) to construct text (programs) which evoke computations to transform input into a desired result. Programs written in BLISS-10 consist of declarations, which establish structure, and expressions sequenced to compute results. Expressions in BLISS-10 can assume remarkably complex forms built up from elementary forms; but regardless of their complexity every expression computes a value. This notion, that a BLISS-10 program consists solely of declarations and expressions, and that these expressions can become arbitrarily complex yet compute a single value during the execution of the program, represents a key concept the reader must understand to properly construct programs in BLISS-10 and fully exploit its power. The concept of a statement prevalent in many programming languages has no meaning in BLISS-10. In reading this manual the reader should strive to master the implications and meaning of the following statement:

BLISS-10 is an expression language.

1.2 EXPRESSIONS

Every executable form in the BLISS language (that is, every form except the declarations) computes a value. Thus all commands are expressions. In the syntax description, E is used as an abbreviation for expression.

```
E ::= smpl-expr/cntrl-expr (1)
```

NOTE

Section 1.3 lists all the simple expressions (smpl-expr) in BLISS-10. Simple expressions generally consist of one or two expressions acted on by an operator. (Note, for example, that 'name' has no operator associated with it but does result in the computation of a value and has a definite place in the precedence hierarchy. Also, the operators FLOAT and FIX are treated as unary operators in BLISS-10 and not as internal functions. The priority of FLOAT and FIX is 5.) This list appears with full knowledge that the reader does not yet know the semantics of many of the operators. It is assumed that the reader is sophisticated and can be depended on to fill in the gaps usually provided by documents with a tutorial intent. The discussion of control expressions (cntrl-expr) appears after discussing a sufficient number of the components making up simple expressions (block, literal, name, etc.) to permit the construction of some simple programs.

----

(1)This specification uses a form similar to BNF (Backus-Naur Form), differing from it by its absence of angle brackets <> to enclose non-terminals in the language. Non-terminals are indicated by the use of lower-case characters. The glossary contains definitions of non-terminals used throughout the specification.

1.3 SIMPLE EXPRESSIONS

The semantics of simple expressions (smpl-expr) is most easily described in terms of the relative precedence of a set of operators, but readers should also refer to the BNF-like description in Appendix A. The precedence number used below should be viewed as an ordinal, so that 1 means first and 2 second in precedence. In the following table the letter E has been used to denote an actual expression of the appropriate syntactic type, (refer to Appendix A).

Table 1-1
Simple Expressions

Precedence	Example	Semantics
1	compound block	The component expressions are evaluated from left to right, and the final value is that of the last component expression.
1	E0(E1,E2,...,EN)	A routine call. (refer to 1.23).
1	E0 [E1,E2,...,EN]	A structure access, (refer to 1.24).
1	name	A pointer to the named item, (refer to 1.4).
1	literal	Value of the converted literal, (refer to 1.10).
2	E<pntr-parms>	A partial word pointer.
5	FIX E	Evaluate E and treat the results as if it were a KA-10 floating point value. The value of FIX E is the corresponding integer equivalent of the value of E.
5	FLOAT E	Evaluate E and treat the result as an integer value. The value of FLOAT E is the KA-10 floating point representation of the value of E.
4	.E	Value (possibly partial word) pointed at by E.
4	@E	Equivalent to .E<0,36,0,0>

(Cont. on next page)

Table 1-1 (Cont)

4	\E	Equivalent to .(t=E)<0,36,.t<18,4>, .t<22,1>> where t is a temporary
5	E1↑E2(1)	E1 shifted arithmetically by E2 bits. Shift is left if E2 is positive, right if negative. Shifts are mod 256.
6	E1*E2	Product of E's.
6	E1/E2 or E1 DIV E2	E1 divided by E2.
6	E1 MOD E2	E1 modulo E2.
7	-E	Negative of E.
7	E1+E2	Sum of E's.
7	E1-E2	Difference between E1 and E2.

NOTE

All arithmetic is carried out modulo 2↑36 with a residue of -2↑35.

6	E1 FMPR E2	Floating multiplication.
6	E1 FDVR E2	Floating division
7	FNEG E	Floating negation
7	E1 FADR E2	Floating addition
7	E1 FSBR E2	Floating subtraction
8	E1 EQL E2	E1 = E2
8	E1 NEQ E2	Not (E1 = E2)
8	E1 LSS E2	E1 < E2
8	E1 LEQ E2	Not (E1 > E2)
8	E1 GTR E2	E1 > E2
8	E1 GEQ E2	Not (E1 < E2)

(Cont on next page)

----
(1)On TTY's without <↑>, a caret will serve as the shift operator.

TABLE 1-1 (Cont.)

9	NOT E	Bitwise complement of E
10	E1 AND E2	Bitwise and of E's
11	E1 OR E2	Bitwise inclusive or of E's
12	E1 XOR E2	Bitwise exclusive or
12	E1 EQV E2	Bitwise equivalence
13	E1 = E2(1)	The value of this expression is identical to that of E2, but as a side effect this value is stored into the partial word pointed to by E1; with associative use of =. The assignments are executed from right to left: thus E1 = E2 = E3 means E1 = (E2 = E3).

----

(1)The store operator is to become <=>. Currently it is <←>. When it becomes equal however, <←> will still be recognized as valid. As a result of the change in the ASCII standard character set, ← appears as underscore on many terminals.

1.3.1 Order of Expression Evaluation(1)

BLISS-10 has two potential side effect producing operations:

1. The assignment operator, and

2. Routine calls.(2)

A side-effect has occurred if the result of expression evaluation depends upon the order in which the expression was evaluated. This can only happen if the same variable appears more than once in the expression and is altered at least once during expression evaluation. For example, the expression

```
(R=2;.R*(R=3))
```

may evaluate to 6 or 9 depending on which of the two expressions

```
.R or (R=3)
```

is evaluated first. Problems of this type are usually resolved by establishing order-of-evaluation rules.

To describe these rules, the five different kinds of ordering in BLISS-10 must be considered:

1. Lexical order (the ordering of program text);

2. Order imposed by operator precedence and parentheses;

3. The order (if any) observed with respect to sub-expressions of commutative operators (e.g. +, and, *, etc);

4. The 'essential' order observed with respect to expressions which produce side effects (specifically, assignments and routine calls).

5. The evaluation rules for expressions whose values may be uniquely determined without evaluating the entire expression.

BLISS-10 applies rules to these cases as follows:

Case 1
The lexical order is determined by the programmer. The one exception is multiple assignment where the order is right to left as in X=Y=Z=A=0;.

----

(1)Though this section logically belongs here, it contains examples that presume knowledge of other sections, in particular 1.4 and 1.5; on a first pass through the manual the readers may find it useful to read these sections before proceeding with 1.3.1.

(2)To avoid needless repetition the terminology 'Routine Calls' will encompass both ROUTINES and FUNCTIONS unless otherwise noted.

Case 2
The effect of operator precedence (hierarchy) and parentheses (grouping), is observed, subject to the definitions in Cases 3 and 4 below,

Case 3
Mathematically commutative operators (+,and,*,etc) are assumed to have computational commutativity as well. Thus E1+E2 may compile as either E1+E2, or E2+E1.

Case 4
The effect of side effects is accounted for only at a semi-colon <;>(1) within a compound expression. Thus the <;> defines a point at which all side-effect producing operations to the left (lexically) of the <;> within the enclosing compound are completed prior to evaluation of expressions to the right of the <;>.

Case 5
An attempt will be made to evaluate an expression only to the point necessary to uniquely determine its value.

Case 5. Exception
Unless a program unconditionally exits a control scope BLISS-10 currently evaluates an entire expression except in the case of constant expressions which evaluate at compile time.

The compiler may make other rearrangements but they are guaranteed to observe the rules stated above.

The evaluation rules defined above permit more code rearrangement than typically found in higher level languages. There are two distinct reasons for this:

1. The compiler can produce considerably better code by allowing it more freedom, and

2. The rules tend to eliminate the use of programming techniques containing 'hidden side effects' since the programmer himself cannot predict the outcome when employing such techniques. Programs with hidden side effects tend to produce code that is difficult to understand and modify, and often generate software faults which defy resolution.

----

(1)Where improved reading flow will result, any angle brackets will enclose non-terminals (the same meta-linguistic device used in original BNF).

The elimination of expressions which produce unpredictable results does require some diligence from the programmer, but the circumstances which give rise to potentially ambiguous results occur rarely in practice, are easily recognized, and their elimination represents good programming practice.

Two programming contexts can produce unwanted side effects, unpredictable results, or both.

Context 1.

1. An assignment statement nested in a larger expression that uses the contents of the variable assigned to in the inner expression, or

2. A routine called with an argument consisting of an undotted name whose contents is altered by the routine but used elsewhere in the expression.

These two situations are manifestations of the same problem but in the second case we consider the potential side effect hidden.

Examples for Context 1

<1> `X=.X*(X=2)`

BLISS-10 can freely determine which of the two components in the simple expression .X*(X=2) it will evaluate first since * is a commutative operator. The outcome clearly depends on the order of evaluation and the programmer must consider it unpredictable.

<2> `X=G(.X)+F(X)+.X`

Here F(X), passed a pointer to X, may change X and if it does the value of entire expression depends on the order of evaluation. Since the + operator is commutative, this construct must be viewed by the programmer as having an unpredictable outcome.

The cautions are simple. Suspect any expressions which

1. Contain multiple store operations on the same variable

2. Contain routine calls containing undotted names of variables appearing elsewhere in the expression.

Context 2

Occurs when a side effect producing sub-expression may not be needed to determine the value of the overall expression.

Examples for Context 2

<1> `X=.A GTR .B AND (C=5) LSS F(B)`

BLISS-10 will perform expression evaluation only to the point at which it can uniquely determine a result. In the above expression both the order-of-evaluation and the outcome can affect the result.

If, for example, .A is in fact less than .B, and if the compiled code evaluated this result first, then F(B) which may change B does not get executed, thus eliminating a possibly assumed side effect.

On the other hand, if the Boolean

```
(C=5) LSS F(B)
```

executes first, and if F(B) changes B then the outcome of the test

```
.A AND .B
```

is clearly affected, possibly unpredictably.

<1> Exception

Currently only expressions with compile time constants conform to this optimization:

```
(.B GTR 0) AND ((.C=.D)LSS 0);
```

will currently make the assignment no matter what the value of .B.

<2> X=0*(Y=1)

Here X will always equal zero (0), but depending on order of evaluation, Y may remain unchanged (since the value of X is always uniquely determined here) or be set equal to 1.

The caution simply exhorts the programmer to beware of expressions whose value can be uniquely determined by the evaluation of a sub-expression contained within it. Failure to do so may result in an assumed side effect not occurring.

1.4 NAMES

Syntactically an identifier, or name, is composed of a sequence of letters and/or digits, the first of which must be a letter. Certain names are reserved as delimiters. Refer to Appendix C. Semantically the occurrence of a name is exactly equivalent to the occurrence of a pointer to the named item. The term "pointer" will take on special connotation later with respect to contiguous sub-fields within a word; however, for the present discussion the term may be equated with "address". This interpretation of name is uniform throughout the language and there is no distinction between left and right hand values. Note that a name belongs to the class of simple expressions, has a position in the precedence hierarchy, and computes a value, namely, the address of the named item.

There is a special case where names may contain the characters "$", "%", and "." as well as letters and digits. This has been provided to be compatible with other DECsystem-10 names, particularly those included in JOBDAT. Any name containing any of these three characters must be immediately preceded by a question mark, and will be delimited by (1) any character other than one of the three, (2) a letter, or (3) a digit. For example,

```
          EXTERNAL ?.JBSYM;

          SYMADDRESS= .?.JBSYM<0,18>;
```

1.5 THE "CONTENTS OF" OPERATORS

Since a name always evaluates to an address, and typically a programmer wants to manipulate the datum stored at the address, the language must provide some notation for accessing the contents of memory. In BLISS-10 this takes the form of the "contents of", or "dot" operator signified by a period <.> prefixing an expression.

The operator <.> is a unary operator used to designate the contents of the location named by its operand. That location may be in core memory or one of the registers. Thus if "X" is the name of a word of memory, then ".X" names its contents, and "..X" names the contents of the word pointed at by the contents of location X.

To further illustrate this basic BLISS-10 concept consider the assignment expression

```
          A=.B
```

In this expression we have two names, A and B, and two operators <.> and <=>. To determine the value of this expression we use the precedence rules of section 1.3. First we derive values for the names (highest precedence). This derives the address of A and the address of B. Then we apply the dot operator to the value of B; thus, we derive the contents of B as the value of .B. Then we apply the store operator, <=>, to the two values thus computed. The store operator semantics store the value on the right hand side into the location pointed to by the value on the left hand side. Algorithmically, A=.B becomes

1. Derive value of B (a pointer);

2. Apply dot to value derived in 1. (contents of B becomes new value);

3. Derive value of A (a pointer);

4. Apply store operator to values derived in 2, and 3. (store contents of B into contents of A).

This simple example illustrates the semantics of two operators (= and .) and how simple expressions build more complex expressions. To carry complexity a step further consider the construction

```
          D=(A=.B)
```

which represents a valid BLISS-10 construction. To determine the semantics, view it as two expressions connected by the store operator; D and (A=.B). The value of an expression involving the store operator takes on the value of the right hand side, so, (A=.B) has a value equal to the contents of B. Thus the entire expression results in storing the contents of B into A and D. Similarly,

```
        A=B=C=D=0
```

will initialize the four variables, A, B, C, and D to 0. (Store operators are left associative). To understand BLISS-10 the reader must master the concept that every BLISS-10 expression computes a value.

NOTE

> BLISS-10 will eventually recognize two characters for the store operators ← and =. This manual, however, uses = exclusively to represent the store operator. Currently it recognizes only ←.

1.5.1 More on "Contents of" Operators and Pointers

The description in 1.5 covers the vast majority of programming requirements and is a machine independent construction. But BLISS-10 has another goal, hardware access, which at times conflicts with that of machine independence. One feature of the DECsystem-10 permits a programmer to pack or unpack bytes(1) anywhere in a word. Movement of a byte is always between an AC and a memory location: a deposit instruction takes a byte from the right end of an AC and inserts it at any desired position in the memory locations; a load instruction takes a byte from any position in the memory location and places it right-justified in an AC.

The byte manipulation instructions have the standard memory reference format, but the effective address E is used to retrieve a pointer, which is used in turn to locate the byte or the place that will receive it. The pointer has the format

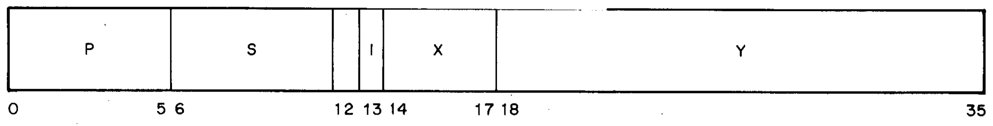


Figure 1-1

Format of a Pointer

---

(1)By byte we mean a contiguous field within a word whose length is between 1 and 36 bits.

where S is the size of the byte as a number of bits, and P is its position as the number of bits remaining at the right of the byte in the word (e.g., if P is 3 the rightmost bit of the byte is bit 32 of the word). The rest of the pointer is interpreted in the same way as in an instruction: I, X and Y are used to calculate the address of the location that is the source or destination of the byte. Thus the pointer aims at a word whose format is

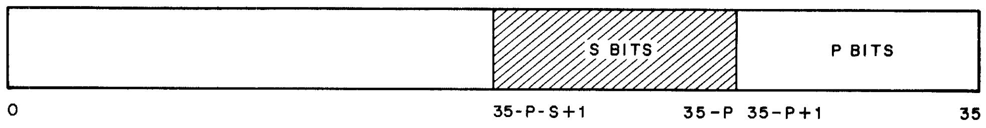


Figure 1-2
Format of a Byte within a Word

where the shaded area is the byte.

The use of these byte pointers can result in coding efficiencies. Indeed, the compiler itself binds every name to a byte pointer with a format identical to that of Figure 1-1. To provide the programmer access to byte pointers, BLISS-10 provides the pointer operator. In its most general form, the pointer operator specifies five quantities which operate on the value of a name to produce a byte pointer as defined in the DECsystem-10 hardware.

Given the expression

```
E<E1,E2,E3,E4>
```

BLISS-10 computes its value as follows

1. Establish the value of E (a 36 bit byte pointer). (If E is a simple name it will normally reduce to an 18 bit Y value with the P, I, and X fields of the byte pointer equal to 0, and the S field equal to 36.)

2. E1 and E2 are computed mod 2↑6 and become the P and S fields of the byte pointer.

3. E3 is computed mod 2↑4 and forms the X field (index field) of the byte pointer.

4. E4 is computed mod 2↑1 and forms the I field (indirect field) of the byte pointer.

The result is a 36-bit value which is, in fact, a DECsystem-10 byte pointer.

Each of the expressions E1, E2, E3, E4, may be omitted with defaults E1, E3, E4=0, and E2=36.

Thus the expression

(A+1)<.B,3>

has a value defined by a byte pointer as follows:

P	S		I	X	Y
.B	3		0	0	A+1

Figure 1-3
Pointer to a 3-bit Field

This byte pointer in fact refers to a 3-bit field in the first location beyond A. The position of this 3-bit field is .B (the contents of B) bits from the right end of the word. (See Figure 1-4.)

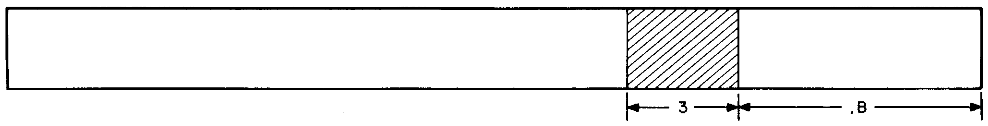


Figure 1-4
A 3-bit Field

Examples:

<1> C=(A+1)<.B,3>

This expression stores the byte pointer of Figure 1-3 into the location pointed to by C.

<2> D=.(A+1)<.B,3>

Store 3 bits .B from the right of A+1 into the field pointed to by D (right justified). Note: names carry a default pointer operator of <0,36>.

<3> F=..C

Where C contains the value from <1>. F now contains the same value as D from example <2>.

The use of byte pointers(1) , though it provides great flexibility, is

----
(1)If the compiler can determine at compile time that a byte pointer points to a fullword or a halfword, it will generate the proper instruction.

inefficient when word or halfword quantities are being manipulated since the DECsystem-10 has explicit instructions to handle words and halfwords. To accommodate the efficiencies inherent in word and halfword operations two additional "contents of" operators are provided, both of which are defined in terms of the dot and pointer operations.

The @ operator produces a fullword pointer ignoring the P, S, X, and I fields. Thus @ is defined as

```
@E=.E<0,36,0,0>
```

To develop the value of the expression on the right.

1. Develop a 36-bit byte pointer value for E.

2. Apply E1=0, E2=36, E3=0, and E4=0 to byte pointer developed in 1, producing a byte pointer with an 18-bit address (Y field).

3. Apply dot, which accesses the location pointed to by the Y field.

The @ operator by eliminating the P and S fields bypasses the need of accessing a byte pointer. (S is eliminated in the sense that it equals 36, implying fullword access.)

The \ operator produces a 23-bit address using I and X but not P and S. It is defined as (t is a temporary)

```
\E==.(t=E)<0,36,.t<18,4>,.t<22,1>>
```

Developing the expression on the right we have:

1. Store E into t (not the contents of E but the 36-bit byte pointer value for E).

2. Set E1=0, E2=36.

3. E3=the X field in the byte pointer of original expression. E4=the I field in the byte pointer of the original expression.

   Note that t contains the original byte pointer value of E. A byte pointer for t with the defined subfields (<18,4>,<22,1>) when applied to the contents of t will retrieve the original E subfields.

4. Using the address (I, X, and Y included) retrieve the contents of memory pointed to.

More Examples:

Suppose the assignment

```
A=B<3,15,R1,0>;
```

has been executed. The assignment stores a byte pointer in A, where P=3, S=15, X=R1, and I=0. Assume R1 contains a 2. Then:

<4> Z=.A

Stores the byte pointer into Z.

<5> Z=..A;

Uses the pointer in A as a byte pointer and stores the field in Figure 1-5 into Z.

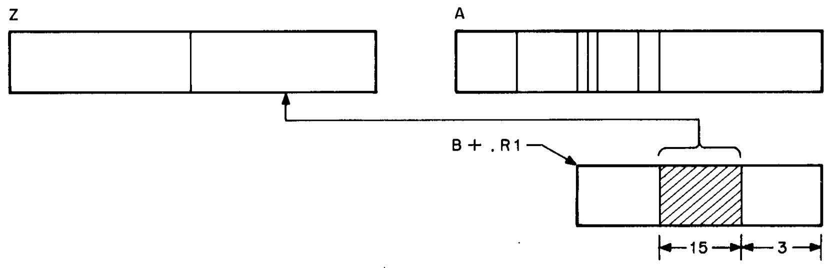


Figure 1-5
Access of a Byte Using a Byte Pointer

<6> Z=@.A;

Stores the contents of B into Z.

Following Figure 1-6 we have:

1. Extract the contents of A.
2. Modify value derived in 1 so that P=I=X=0,S=36.
3. Interpret the result of 2 as a pointer.
4. Extract the pointed-to word and store it into Z.

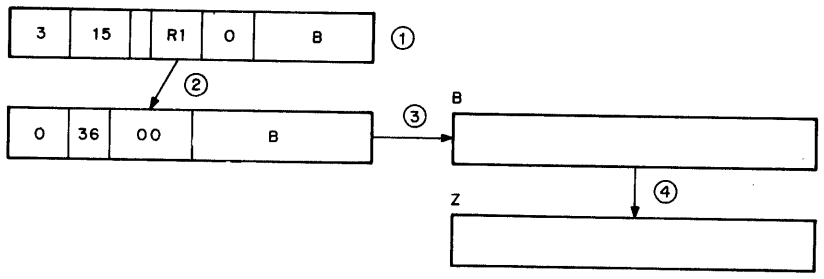


Figure 1-6
Using the   Operator to Modify Use of a Byte Pointer

<7> Z=\.A;

Store the contents of the second word following B into Z.

Following Figure 1-7 we have:

1. Extract the contents of A.
2. Modify value derived in 1 so that P=0, S=36.
3. Interpret the result in 2 as a pointer.
4. Extract the pointed-to word and store it in Z.

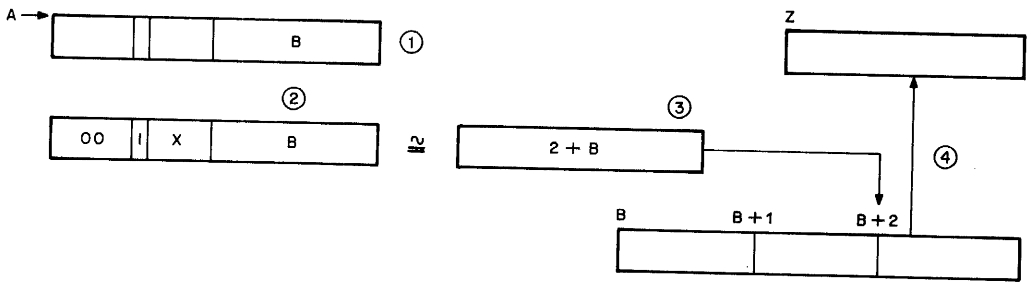


Figure 1-7
Using the \ Operator to Modify Use of a Byte Pointer

<8> .A=5;

Stores 5 (right-justified, zero-filled) into the 15-bit field in the second word following B as shown in Figure 1-8.

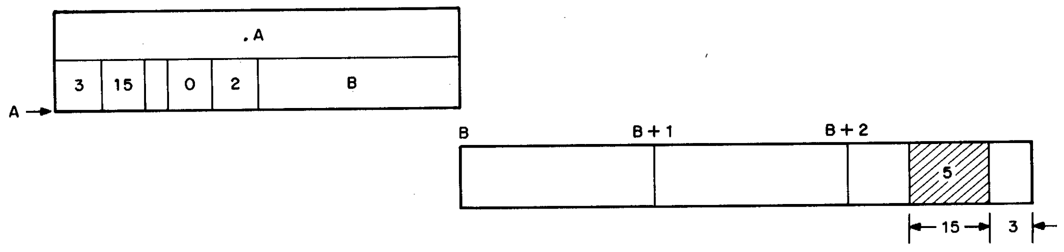


Figure 1-8
Assignment to a Byte Using a Byte Pointer

<9> @A=5;
.A=5;
\A=5
are equivalent.

@A=5
decomposes as

1. Form a 36-bit byte pointer form A

2. Apply @ to this value, which effectively leaves it unaltered.

3. Take the value pointed to by the value derived in 2 (the contents of A) use this as a pointer and store 5 in the pointed-to word.

Both
.A=5
and
\A=5

will produce exactly the same result, because A, a name, is bound to a byte-pointer (which is its value) such that both @ and \ leave this value unchanged.

<10> @.A=5
and
(.A)<0,36>=5
are not equivalent.

Following Figure 1-9 @.A=5 decomposes as

1. Extract contents of A, set P=0, S=36, X=0, I=0.

2. Use the value from 1 as a pointer; extract the word pointed to.

3. Using the value in 2 as a pointer store 5 in the pointed-to word.

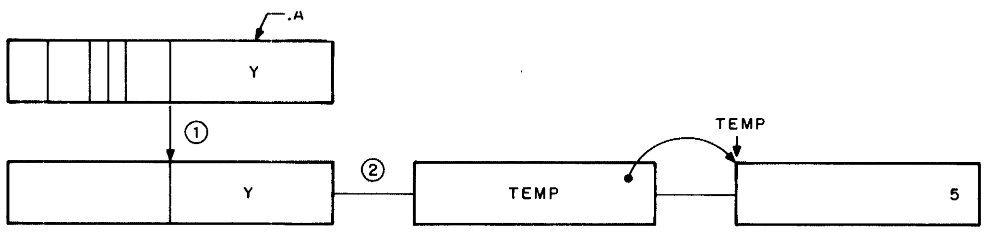


Figure 1-9
Description of @.A=5

Following Figure 1-10

(.A)<0,36> decomposes as:

1. Extract contents of A, set P=0, S=36, X=0, I=0.

2. Using value derived in 1 store 5 into word pointed to.

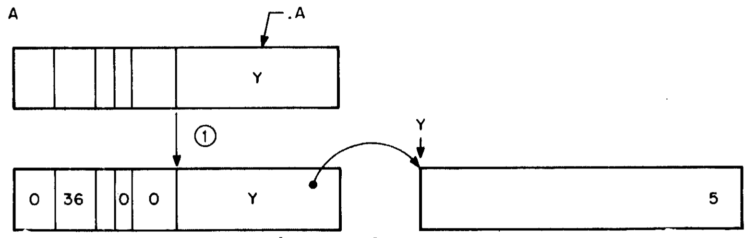


Figure 1-10
Description of (.A)<0,36>=5

1.6 DATA REPRESENTATION

In general, all values are treated as strings of bits. Different operators will interpret these strings in various ways; for instance, (1) arithmetic operators (+,-,etc.) interpret values as 2's complement integers, (2) floating point operators (FADR, etc) interpret values as KA-10 floating point values, (3) logical operators (AND, OR, etc.) interpret values as bit strings, and (4) access operators (<, ., >) interpret values as byte pointers.

All Boolean tests depend on the low-order bit of the word or field being tested. The relational operators generate a full word 0 or 1 as their value, when required to yield a value. Zero represents falsity, and one represents truth.

1.7 COMMENTS

Comments may be enclosed between the symbol ! and the end of the line on which the ! appears or between paired % symbols. Note that a ! or % enclosed within quotes is part of the quoted string, not the start of a comment. Also a % after ! and before the end of the line, or ! between %'s is part of the enclosing comment.

```
          comment ::= ! restofline end-in-sym/
                      % strng-no-pct %/empty
```

Examples:

```
          !Start a comment
          %Hurrah for Karamazov!%
          !the GNP declined 3%
```

The compiler will not accept a line of source or comment which contains over 135 characters. In BLISS-10 a line is terminated by a carriage return. On input all source characters with ASCII codes less than #40 are ignored except for tab (#11), carriage return (#15), and form feed (#14).

1.8 MODULES

A module is a program element which may be compiled independently of other elements and subsequently loaded with them to form a complete program.

```
          module      ::= block/
                          module-head block/
                          module-head block ELUDOM/block ELUDOM
          module-head::= MODULE name (mdle-parms)=
```

A module may request access to variables and functions declared in other modules by declaring their names in EXTERNAL declarations. A module permits general use of its OWN variables and ROUTINES by means of GLOBAL declarations. These lines of communication between modules are completed by the loader prior to execution. A complete program consists of a set of complied modules bound by the loader. The <name> in a module declaration is used to identify that module and must be unique in its first four characters from any other module names which are to be loaded together to form a complete program. When loading a number of modules, it is necessary that at least one module contain a module head. The use of the keyword MODULE is optional but if no module appears, BLISS-10 will default the name to that of the REL file requested.

A terminating ELUDOM is optional and is ignored whether or not a module has a module-head.

The (mdle-parms) field of a module definition is used to control the compilation. Refer to section 3.3-3.5 for a descriptive list of parameters and defaults.

Example:

```
          !The start module for BLISS is compiled in front
          !of a number of other BLISS modules.

          MODULE START (STACK=EXTERNAL(STACK,#2000)=
          BEGIN
              .
              .
              .
          END
          ELUDOM
```

1.9 BLOCKS AND COMPOUND EXPRESSIONS

A block is one or more declarations followed by an arbitrary number of expressions all separated by semicolons and enclosed in a matching BEGIN-END or "(" - ")" pair.

```
          block ::= BEGIN blockbody END / ( blockbody )
          blockbody ::= decls exprs
          decls ::= decl;/decls; decl
          cmpnd-expr ::= BEGIN exprs END /( exprs)
          exprs ::= E/label:E/ E; exprs/
                    E SEMICOLON exprs/empty(1)
```

The block indicates the lexical scope of the names declared at its head. However, names of variables and ROUTINES declared as GLOBAL have a scope beyond the block although they are declared within the module. The effect, for a module citing them in an EXTERNAL declaration, is as if they were declared in the current block.

This violation of conventional block structure has implications with respect to allowed references, particularly in connection with declared registers. These implications, and a corresponding set of restrictions, will be discussed in connection with the affected declarations.

The reserved identifier SEMICOLON is identical to the character ";" syntactically. In addition, it is a directive to the compiler declaring that the expression just completed may have side-effects which are unpredictable by the compiler. Consequently, no assumptions should be made about valid temporary or intermediate results for optimization purposes.

----

(1)The string <empty> will mean that an option for the construct in question may be empty. For example, a block may consist of nothing but declarations.

1.10 LITERALS

The basic data element on the DECsystem-10 is the 36-bit word. The hardware, however, permits access to an arbitrary length contiguous field within a word. The programmer can view the 36-bit word as the limiting case of a partial word. Refer to section 1.5.1 for additional information on partial words.

```
literal ::= number/string/plit
string ::= string-type quoted-string
strng-type ::= ASCII/ASCIZ/RADIX50/SIXBIT/empty
quoted-string ::= leftadjstring/rightadjstring
leftadjstring ::= 'strng-no-sngle-''
rightadstring ::= "strng-no-sngle-""
number ::= decimal/octal/floating
decimal ::= digit/decimal digit
floating ::= decimal.decimal/
             decimal.decimal exponent/
             decimal exponent
exponent ::= E decimal(1)/E+decimal/E-decimal
octal ::= #oit/octal oit
oit ::= 0/1/2/3/4/5/6/7
digit ::= 0/1/2/3/4/5/6/7/8/9
```

Numbers (unsigned integers) are converted to binary modulo 2(36) residue -2(35). The binary number is 2's complement and is signed. Octal constants are prefixed by the sharp sign <#>. Quoted-string literals may be used to specify bit patterns corresponding to the 7-bit ASCII code for printing graphic characters on the external I/O media. Strings of one to five characters may be used freely as character constants.

BLISS-10 will allocate the specified data in increasing memory address order.

The ASCII string-type converts the quoted-string following the keyword ASCII into five 7-bit ASCII characters. ASCIZ creates an ASCII string terminated with at least one zero byte; for strings containing an even multiple of five characters, a zero word.

RADIX50 packs the six characters into 32 bits right justified with the left 4 bits equal to 0. A quoted-string following RADIX50 cannot contain more than six characters from the set <A-Z>,<0-9>,<$>,<.>,<%>, and <null>.

Quoted-string literals may be used to specify bit patterns corresponding to the ASCII, SIXBIT, or RADIX50 codes used on the DECsystem-10; left or right justification may be obtained through the use of the single or double quote chracters. An empty 'stringtype' implies an ASCII string. Normal quoted strings are constrained to be representable within a single word (five characters for ASCII, six characters for SIXBIT and RADIX50), but strings of arbitrary length may be used in PLIT's (refer to 1.11).

----
(1)This is the literal E, not an expression.

Within a quoted string the quoting character is represented by two successive occurrences of that character. Also, in an ASCII or ASCIZ quoted string the question mark, <?>, is an escape-to-control character -- thus <?M> represents a <control-M>, or carriage return. In addition

```
          <??> represents a question mark itself
          <?0> represents the NULL (zero) character
          <?1> represents the DEL (all ones) character
```

1.11 POINTERS TO LITERALS

A PLIT is a pointer to a literal word whose contents are specified at compile time and established at load time; e.g., PLIT 3 is a pointer to a word whose contents will be set to 3 at load time.

```
          plit ::= PLIT plitarg
          plitarg ::= load-time-expr /
                      long-string /
                      triple
          triple ::= (triple-item-lst)
          triple-item-lst ::= triple-item /
                              triple-item,triple-item-lst
          triple-item ::= load-time-expr /
                          long-string /
                          dup-fctr:plitarg
          dup-fctr ::= compl-time-expr
```

NOTE

"PLIT (3)+4" has 2 parses:
PLIT load-time-expr, and
PLIT triple + expr

The latter choice is used. Hence, "PLIT (3)+4" is the same as "(PLIT 3)+4". Note that PLIT (3)+4 yields a value that may have little or no meaning. PLIT (3) produces a pointer to a memory location containing a 3. Adding 4 to this pointer may not produce the intended value.

A PLIT may point to a contiguously stored sequence of literals; long strings and nested lists of literals are also allowed. The value of

```
          PLIT (3,5,7,9)
```

is a pointer to four contiguous words containing 3,5,7 and 9, respectively. A long string (more than 5 characters) is also a valid argument to a PLIT:

```
          PLIT "this allocates 5 words"
```

allocates five words of 7-bit ASCII characters with three pad characters of zero to the right.

Note: A long string cannot exceed 1,000 characters in the current implemenation of BLISS-10.

The arguments to PLITs need only be constant at load time; PLITs are themselves literals, thus nesting of PLITs is allowed (with the inner PLITs allocated first):

Name binding is possible within a PLIT by use of the NAMES or INDEXES facility.

EXAMPLE:

```
BIND APLIT=PLIT(
           NAME1 GLOBALLY NAMES 1,
           INDEX2 GLOBALLY INDEXES NAME2 NAMES 2,
               3);
```

In this example the identifier NAME1 is bound to the address of the PLIT element which it precedes (i.e., 1). Use of the word GLOBALLY makes this identifier available as a global symbol to separately compiled modules. The identifier INDEX2 is bound to the offset of the element it precedes in the PLIT.

Thus the following produce equivalent results, even though some of the names are not valid in all scopes. Note that in BLISS-10 all indices start at 0, not at 1.

```
          X=.NAME1[1];
          X=.NAME1[INDEX2];
          X=.APLIT[1];
          X=.APLIT[INDEX2];
          X=.NAME2;
```

Name binding of this sort can occur in nested PLITS. Name binding cannot occur in a portion of a PLIT subject to a duplication factor. In this case a warning message is generated and the name binding is ignored.

Any triple-item may be preceded by any number of occurrences of a plit-name-bind where

```
          plit-name-bind::=name NAMES/
                           name GLOBALLY NAMES/
                           name INDEXES/
                           name GLOBALLY INDEXES
```

The following example introduces two declarations; EXTERNAL and BIND. EXTERNAL declares that the names following it exist in an independently compiled module (thus evaluatable only at load time). BIND establishes an equivalence between the variable name on the left of the equal sign, =, and the expression on the right, such that when the variable name appears in the program the compiler substitutes the value of the expression for the variable name. The use of the equal sign does not indicate a store operation in the BIND context, merely association between the name and the expression (see section 1.22.2 for details on EXTERNAL and 1.27 for details on BIND).

Examples:

```
EXTERNAL A,B,C:
BIND Y = PLIT (A, PLIT (B,C), PLIT 3, 'A LONG STRING', 5+9*3);
```

is such that (see Figure 1-11):

```
 .y[0]==A<0,36>;..y[1]==B<0,36>;.(.y[1]+1)==C<0,36>
..y[2]==3;.y[3]=="A LON";.y[4]=="G STR";.y[5]=="ING";
 .y[6]==32;
```

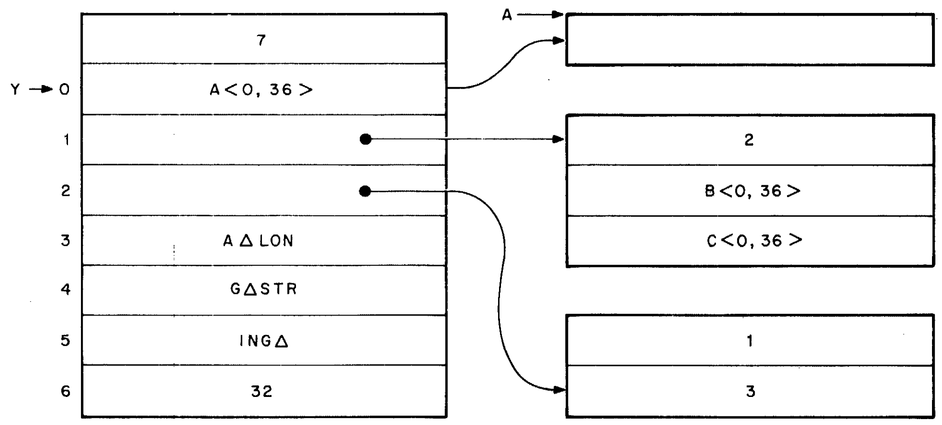


Figure 1-11
Memory Allocation of some PLITs

The notation Y[i] can be viewed as vector addressing with Y[0] pointing to the first element of the vector. Actually this notation involves a BLISS-10 structure access which is discussed in sections 1.23 through 1.25.

In addition, any argument to a PLIT can be replicated by specifying the number of times it is to be repeated; e.g.,

```
PLIT (7:3)
```

produces a pointer to 7 contiguous words, each of which contains the value 3, duplicated PLITs are allocated once, identical PLITs are not pooled. Hence,

```
BIND X = PLIT (3: PLIT A, PLIT A, 2: (2,3));
```

is such that:

```
..x[0] == ..x[1] == ..x[2] == ..x[3] == A<0,36>;(1)
 .x[0] == .x[1]  ==  .x[2] not == .x[3];
 .x[4] == .x[6]  ==  2; .x[5] == .x[7] == 3;
```

.X[0],.X[1], .X[2] are equal since three literal pointers are generated all pointing to a single instance of A. .X[3] does not equal .X[0], .X[1], .X[2] because it results in a separate allocation for A. Figure 1-12 shows the entire PLIT generation.

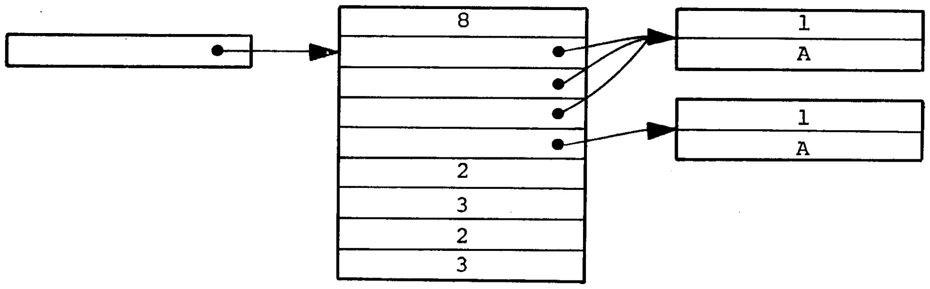


Figure 1-12
Memory Allocation of some PLITs with Replication Factors

Note: The length of every PLIT (in words) is stored as the word preceding the PLIT. Hence, in the last example, .X[-1] = 8.

A replication factor of zero(0) results in no allocation.

1.12 INTERLUDE 1

In Section 1.2 we began searching the basic building blocks of BLISS-10 programs namely, expressions, but interrupted it exposing the reader to names, the dot and store operators, modules, blocks, data representation, literals, pointers to literals (plits), and pointers. We now return to the discussion of expressions. It may prove useful for the reader to review Sections 1.1 and 1.2 before proceeding.

----

(1)Note the use of two successive equal signs (==) means equivalence. We use this convention as a pedagogical mechanism to avoid confusion with the store operator <=>. BLISS-10 has no syntactical structure using <==>.

1.13 CONTROL EXPRESSIONS

The control expressions provide the mechanism needed for controlling the execution sequence of a program. BLISS-10 has five forms:

```
          cntrl-expr::=cndlt-expr/
                       loop-expr/
                       choice-expr/
                       escp-expr/
                       co-rtn-expr
```

1.14 CONDITIONAL EXPRESSIONS

```
          cndtl-expr ::= If E1 THEN E2 ELSE E3;
```

E1 is computed and the resulting value is tested. If it is true, then E2 is evaluated to provide the value of the conditional expression, otherwise E3 is evaluated to provide the value of the conditional expression.

```
          cndtl-expr ::=IF E1 THEN E2
```

This form is equivalent to if E1 then E2 else 0. However, it does introduce the "dangling else" ambiguity. This is resolved by matching each ELSE to the most recent unmatched THEN as the conditional expression is scanned from left to right.

Examples:

```
<1>(1) IF .X THEN J = .K ELSE J=.L;

<2>  J = (IF .X THEN .K ELSE .L);!Same effect as
                                 !previous line

<3>  IF .L THEN
             BEGIN.....END
                ELSE
             BEGIN.....END
               ;                    !Blocks allow multiple
                                    !expressions

<4>  position = .position + (IF .char EQL #11 %tab% THEN
     8 ELSE 1);

     !Note the use of an octal literal, lower case names,
     !and the comment enclosed in % symbols (#11
     !represents the octal code for tab.)
```

----

(1)The single integer values in angle brackets simply number the examples, and do not belong to the syntax of the language.

Another form of the conditional expression is:

```
          IFSKIP E1 THEN E2 ELSE E3;
```

In this case, E1 is evaluated first. If the last instruction in E1 causes a skip, (including a routine call with no parameters), then E2 is evaluated to provide the value of the expression, otherwise E3 is evaluated.

Example:

```
        MACHOP  TTCALL=#51;    MACRO SKPINC=TTCALL(#13,0)$;

        IFSKIP SKPINC THEN     !IS A CHARACTER IN THE INPUT
                               BUFFER?
             BEGIN ... END     !YES
               ELSE
             BEGIN ... END     !NO
               ;
```

1.15 LOOP EXPRESSIONS

The value of each of the six loop expressions is -1, except when an EXITLOOP or LEAVE is used, see 1.16.

```
          loop-expr ::=WHILE E1 DO E2
```

The E1 is computed and the resulting value is tested. If it is true, then E2 is computed and the complete loop-expr is recomputed; if it is false, then the loop-expr evaluation is complete.(1)

```
          loop-expr::=UNTIL E3 DO E2
```

This form is equivalent to WHILE NOT(E3) DO E2

```
          loop-expr:=DO E2 WHILE E1
```

The expressions E2, E1 are computed in that sequence. The value resulting from E1 is tested: if it is true, then the complete loop-expr is recomputed: if it is false, then the loop-expression evaluation is complete.

```
          loop-expr::=DO E2 UNTIL E3
```

This form is equivalent to DO E2 WHILE NOT (E3)

```
     loop-expr::=INCR name FROM E1 TO E2 BY E3 DO E4
```

The <name> is declared to be a register or a local for the scope of the loop. The expression E1 is computed and stored in <name>. The expressions E2 and E3 are computed and stored in unnamed local memory which for explanation purposes we shall name U2 and U3, any of the phrases FROM E1, TO E2, or BY E3 may be omitted--in which case default values of E1 = 0, E2 = 2↑35-1, E3 = 1 are supplied. The effect is as if the following loop-expr were executed:

```
     BEGIN REGISTER NAME;  LOCAL U2,U3; NAME=E1; U2=E2; U3=E3;(2)
     UNTIL .NAME GTR .U2 DO (E4; NAME=.NAME+.U3)
     END
```

----

(1)Only the low order bit is tested; 0= false 1= true.

(2) Section 1.25 discusses the declarations REGISTER and LOCAL.

The final form of a loop-expr is:

```
    loop-expr::=DECR name FROM E1 TO E2 BY E3 DO E4
```

This is equivalent to INCR name FROM E1 TO E2 BY -(E3) DO E4.

CAUTION:
BLISS-10 uses signed relationals in the loop expressions. This can cause unexpected results if any of the loop variables represent pointers rather than numerical values.

If any of the FROM, TO, or BY phrases are omitted from a DECR expression, default values of E1=0, E2=-2↑35, and E3=1 are supplied. Notice that in both forms the end condition is tested before the loop, hence the loop is potentially executed zero or more times.

Examples:

```
<1>  !Following algorithm will place the address of the
     !last item of a linked list into the variable
     !LINK. Figure 1-13 shows storage layout initially

     LINK=.LISTHEAD;
     WHILE ..LINK NEQ 0 DO
        BEGIN
          LINK=..LINK
        END;
```

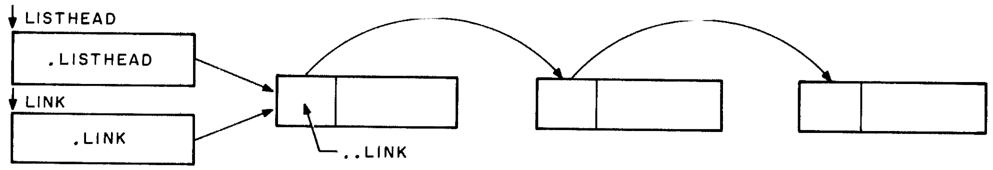


Figure 1-13
Computing the Address of the Last Item in a Linked List

```
<2>  !Add up the first n integers
     !N contains ordinal count of integers required
     !in the sum
     SUM=0;
     INCR J FROM 1 TO .N DO SUM=.SUM+.J
```

1.16 ESCAPE EXPRESSIONS

The escape expressions permit control to leave its current environment. There are three forms:

```
          esc-expr::=LEAVE label WITH E/
                     RETURN E/
                     LEAVE label/
                      RETURN
```

Any expression may be labeled by preceding it with the label name and a colon. Within a labeled expression, control may be caused to leave the expression and yield E as the value of the expression.

A LEAVE expression must occur within an expression with the same label. Refer to 1.28 for restrictions on the use of labels.

If the WITH E is missing then WITH 0 is presumed.

RETURN exits the currently executing routine, yielding E as the value of the routine.

Examples:

```
<1>   !Find INDEX of first space in line image of 80
      !characters
      !INDEX is -1 if none found because the value of an
      !exhausted loop-exp equals -1.
      INDEX=LOOP:INCR J FROM 0 TO 79 DO IF .(LINE+.J) EQL #40
                     THEN LEAVE LOOP WITH .J

<2>   !How to escape to next iteration in a loop
      INCR J FROM 1 TO 100 DO
           L2:(...
               ...
               IF .condition THEN LEAVE L2;
                                           !Exit the compound
                                           !which
               ...                         !constitutes the loop
                                           !body. The exit
              );                           !terminates the
                                           !current iteration
                                           !immediately causing
                                           !the next iteration
                                           !of the DO loop.

<3>   !Find first zero element of a 2-D array
       L3:BEGIN INCR I FROM 1 TO .IMAX DO
           INCR J FROM 1 TO .JMAX DO
              IF .ARRAY[.I,.J] EQL 0 THEN (II=.I;JJ=.J; LEAVE
              L3)
          END

      !This example results in complete loop termination
      !since the label, L3, labels the BEGIN INCR...END
      !expression. Contrast with preceding example.
```

1.16.1 Exception

Escape Expressions That Will Disappear

The following forms of escape expressions in addition to those of Section 1.16 permit control to leave its current environment. These forms will soon be obsolete and the programmer should avoid their use.

```
escpe-expr::=envt level escp-val
envt::=EXIT/EXITBLOCK/EXITCOMPOUND/EXITLOOP/EXITCOND
        EXITCASE/EXITSET/EXITSELECT/BREAK/
        EXITCOMP/EXITCONDIT
level::=[E]/empty
escp-val::=E/empty
```

Each of these expressions conveys to its new environment a value, say E, obtained by evaluating the escape value (escp-val), which may optionally be omitted implying E=0. The levels field, which must evaluate to a constant, say n, at compile time, determines the number of levels of the specified control environment to be exited; the levels field may optionally be omitted in which case one level is implied. The maximum number of levels which may be exited in this way is limited by the current function (routine) body or the outermost block.

EXITBLOCK	terminates the innermost n (where n is the value of the "levels" field) blocks, yielding a value of E for the outermost one exited.
EXITCOMPOUND or EXITCOMP	terminates the innermost n compound expressions, yielding a value of E for the outermost one exited.
EXITLOOP or BREAK	terminates the innermost n loop expressions, yielding a value of E for the outermost one exited.
EXITCOND or EXITCONDIT	Terminates the innermost n conditional expressions, yielding a value of E for the outermost one exited.
EXIT	terminates the innermost n control scopes (whether blocks, compounds, conditionals, or loops) with E as the value of the outermost.
EXITCASE	terminates the n innermost case expressions yielding a value of E for the outermost of these.
EXITSET	terminates the n innermost set expressions in a case expression, yielding a value of E for the outermost of these.
EXITSELECT	terminates the n innermost select expressions yielding a value of E for the outermost of these.

Examples:

```
! find INDEX of first space in line image of 80 characters
(one per word)

! INDEX = -1 implies none found

index = incr j from 0 to 79 do
          if .(line + .j) eql #40 then exitloop j;
```

```
! to exit the body of a loop and go on to the next iteration
INCR J FROM 1 TO 100 DO
  ( ...
  ...
  IF .condition THEN EXITCOMPOUND; !go to next iteration
  ...
  )                              !close compound
  ;                              !close body
                                  expression
```

1.17 CHOICE EXPRESSIONS

1.17.1 CASE Expression

```
choice-expr::=CASE expr-lst OF SET expr-set TES(1)
expr-lst   ::=E/E,expr-lst
expr-set   ::= E/;expr-set/E;expr-set/empty
```

Consider each expression in expr-lst to be numbered, i.e., E0, E1,...; and each expression in expr-set similarly, i.e., ES0, ES1, ... Then for each En, n running from 0 until expr-lst is exhausted, the En is evaluated, yielding a value m. This value is used to select a member of expr-set, namely ESm, which is evaluated. The value of the CASE expression is the value of the last ESm evaluated.

The following points should be noted:

1. A value of -1 for any En terminates the case expression.
2. A value for any En less than -1 or greater than the number of expressions in the expr-lst can cause unpredictable results. No range checking is performed.
3. If the CASE expression is prematurely terminated (see 1), its value is undefined if no ESm has been evaluated.

----
(1)BLISS-10 uses the convention of forward-reverse spelling of keyword delimiters in every case except BEGIN-END.

Examples:

```
<1>  !Suppose TYPE(.CHAR)=0(1) if .CHAR is a number, 1 if a
     !letter, 2 if ignorable: equal to space, tab, and 3
     !otherwise. Collect a string of letters and digits, with
     !COUNT being the length
     !ID names the location used to collect successive
     !characters in the identifier. The algorithm
     !first checks for a character and if found begins
     !to collect the identifier in ID.
     !NXTCHR is a routine with no parameters which
     !returns as its value the next character.

     COUNT=0;
     L3:IF TYPE(.CHAR) EQL 1 THEN
       DO
         CASE TYPE(.CHAR) OF
           SET
           ((ID+.COUNT)=.CHAR;COUNT=.COUNT+1); !CASE=0
           ((ID+.COUNT)=.CHAR;COUNT=.COUNT+1); !CASE=1
           0;                                  !CASE=2
           LEAVE L3                            !CASE=3
           TES
       WHILE(CHAR=NXTCHAR();1);

     !CASE 0 deposits a digit in ID(properly indexed by
     !.COUNT) and increments COUNT.
     !CASE1 does the same for a letter.
     !CASE2 bypasses the character.
     !CASE3 terminates the scan.
     !The WHILE loop causes an infinite loop since
     !it always has the value TRUE; escape via
     !LEAVE L3
```

1.17.2 SELECT Expression

The SELECT expression is similar to the CASE expression in that it selects from the expressions between the NSET-TESN brackets, depending on the values of the expr-lst. The expressions in the d-expr-set are not thought of as being ordered -- each element consists of an "activation expression" tagging a "target expression". The criterion for evaluation of the target expression is that the value of the activation expression is equal to the value of (at least) one of the elements in the expr-lst.

Evaluation of the SELECT expression proceeds as follows: First, the expressions in the expr-lst are evaluated and their values saved. In an undefined order, except when defined by the use of OTHERWISE or

----

(1)TYPE(.CHAR) results in a call on the subroutine TYPE which returns the values defined. The discussion of ROUTINES appears in Section 1.22.

ALWAYS, each activation expression is evaluated and its value compared to the saved values of expr-lst. If any of these values is equal to the value of the activation expression, the target expression is evaluated. The value of the SELECT expression is the value of the last target expression evaluated, or -1 if no target expression was evaluated.

For example, the evaluation of:

```
     SELECT E1, E2, E3 OF
          NSET
          E4: E5 ;
          E6: E7 ;
          E8: E9 ;
          TESN
```

proceeds as follows: E1, E2 and E3 are evaluated. Then E4 is evaluated. If the value of E4 equals either of E1, E2, or E3, then E5 is evaluated. Similarly, E6 is evaluated and tested, possibly causing evaluation of E7; E8 and E9 follow. The value of the expression is the last target expression (E5, E7, or E9) to be evaluated; or -1 if neither E5, E7 nor E9 were evaluated.

There are two reserved words which may be used as activation expressions - OTHERWISE and ALWAYS. ALWAYS causes the target expression to be evaluated whatever the values in the expr-lst. OTHERWISE causes the target expression to be evaluated if and only if no target expression has been previously evaluated in this SELECT expression. A semicolon missing before an OTHERWISE or ALWAYS will generate a warning message, and be assumed present.

Note that although ALWAYS or OTHERWISE may be used as any activation expression, it makes no sense to use more than one OTHERWISE or to use an OTHERWISE after an ALWAYS, since the latter OTHERWISE can never cause the target expression to be evaluated.

1.17.3 Compile-time Constants in Conditional and Choice Expression

Many BLISS-10 constants require that non-terminals become constants at compile time. For example the declaration

```
OWN X[E]
```

requires that E be completely determined at compile time. As long as E does reduce to a compile-time constant, it can admit to expressions the programmer would not normally consider in the context of a compile-time constant. In particular we may write without generating an error

```
OWN X[IF E1 THEN E2 ELSE E3]
```

if we can guarantee that the control expression will reduce to a compile time constant. BLISS-10 permits control-expressions and CASE expressions to be used in this manner, but not SELECT expressions. Use of the control and CASE expressions in this manner can prove quite useful in effecting conditional compilations and as a technique for simplifying the achievement of algorithm and data independence.

1.18 CO-ROUTINE EXPRESSIONS

Coroutines can often simplify a given problem both in its conception and its implementation. (See references on next page.) Coroutines differ from subroutines in that subroutines are always initiated at the beginning while coroutines are always initiated at the location following where they left off. Getting coroutines started is a matter of properly initializing the linkage. Each coroutine has a return address and one (that of the last coroutine relinquishing control) must be preserved when control is passed.

In BLISS-10, the body of a function or routine may be activated as a co-routine and/or asynchronous process; the additional syntax is

```
          co-rtn-expr::=CREATE E1 (expr-lst) AT E2 LENGTH
                        E3 THEN E4/
                        EXCHJ (E6,E7)
```

The effect of a 'create' expression is to create a context (i.e., an independent stack) for the routine (function) named by E1, with parameters specified by the expr-lst, at the location whose address is specified by E2, and of size E3 words. The value of a 'create' expression is the address specified by E2. Control then passes to the statements following the 'create'. When two or more such contexts have been established, control may be passed from any one to any other by executing an exchange-jump, EXCHJ (E6,E7)(1), where the value of E6 must be the stack base, E2, of a previous 'create' expression. The value of E7 is made available to the called routine as the value of its own EXCHJ which caused control to pass out of that routine. Thus

----

References

1. Knuth, Donald
   The Art of Computer Programming
   pps. 210-250
2. Maurer, Ward
   Programming Languages
   pps. 25-50
3. Conway, M. E.
   Design of a Separable Transition-Diagram Compiler
   Communications of the ACM July 1963, pps. 396-407

----

(1) Note that the 1st EXCHJ to a newly created process causes control to enter from its head with actual parameters as set up by the CREATE.

(2)The value of E7 is not available to the called routine on the 1st EXCHJ to it.

the value of the EXCHJ operation is defined dynamically by the co-routine which at some later time re-activates execution of the current co-routine.(2)

Initialization of the main BLISS program is, in fact, equivalent to a CREATE followed by an EXCHJ with E6 being the STACK parameter as specified in the module head.

Should a process, the body of which is necessarily that of a routine (or function), execute a 'return', either explicitly or implicitly, the expression E4 (following the 'then' in the 'create' expression of the creating process) is executed in the context of the created process. The normal responsibilities of E4 include making the stack space used for the created context available for other uses and performing an EXCHJ to some other process.

The facilities described above, namely 'CREATE' and 'EXCHJ', are adequate either for use directly as co-routine linkages or for use as primitives in constructing more sophisticated co-routine facilities with macros and/or procedures. It should be noted in the context that if the created processes are functions (rather than routines) the resulting processes continue to have access to lexically global variables which may be local to an embracing function (access to lexically local variables which have been declared 'own' is available in either case). In such a case the resulting structure is a stack tree in which all segments of the tree below the lexical level of the (function) process are available to it.

Two additional complexities are added if the CREATE and EXCHJ are to be used for asynchronous, and possibly parallel, execution of processes. One is synchronization, by which we mean a mechanism by which a process can coordinate its execution with that of one or more others. A typical example of the need for synchronization occurs when two processes independently update a common data base; and each must be sure that the entire updating process is complete before any other process attempts to use the data base. The second complexity arises in connection with interrupts, and in particular from the fact that certain operations must not be interrupted (some EXCHJ operations for example). It is possible that certain situations require synchronization mechanisms but do not need to be concerned about the interrupt problem--as for example, a user program with asynchronous processes, which is 'blind' to interrupts, and which some monitor systems view as a single 'job'.

The nature of "appropriate" synchronization primitives and mechanisms for temporarily blinding the processor to interrupts (or interrupts in a certain class) are highly dependent upon the nature of the processes being used and the operating system, or lack of one, underlying the BLISS-10 program. As a consequence, no syntax for dealing with either problem is included in the language; in any case, the amount of code necessary for these facilities is quite small.

The co-routine user is well advised to read and understand the material on the run-time representation of BLISS-10 programs contained in Chapter 4.

Example

<1> The coroutine mechanism described above is illustrated by the following skeletal example.

```
             BEGIN
               OWN PA,PB,S1[100],S2[100];
               FUNCTION A=
                 BEGIN LOCAL LA,X;
                 .
                 .
     4.          X=EXCHJ(PB,LA);
                 .
                 .
                 .
                 END;
               FUNCTION B(Z)=
                 BEGIN LOCAL LB,Y;
                 .
                 .
                 .
     5.          Y=EXCHJ(.Z,LB);
                 .
                 .
                 .
                 END;
     1.      PA=CREATE A() AT S1 LENGTH 100 THEN EXIT;
     2.      PB=CREATE B(.PA) AT S2 LENGTH 100 THEN EXIT;
     3.      EXCHJ(.PA,0);
           END
```

Execution flow (corresponding to statement numbering)

1. Create a coroutine context for the function A, and store a pointer to this context in PA.

2. Create a coroutine context for the function B and initialize its one formal parameter to a pointer to an incarnation of A. Store a pointer to the created context in PB.

3. Start A; the E6 parameter, in this case 0, is discarded on the initial activation of a coroutine.

4. A is suspended as a result of the EXCHJ at the point just prior to the store operation. This suspension is exactly the semantics if the EXCHJ were a function call. The EXCHJ, like a function, has a value; its value equals the value returned on reactivation of the coroutine. B is activated, and on its first activation, the value LA is discarded.

5. B now reactivates A (.Z is a pointer to A). Suspension of B occurs prior to the store. On the reactivation of A, the value LB is stored into X in A.

1.19 DECLARATIONS

All declarations, except MAP and SWITCH, introduce names, each of which is unique to the block in which the declaration appears; and with two exceptions (i.e., STRUCTURE and MACRO declarations), the name introduced has a pointer bound to it.

The declarations are:

```
decl::=routn-decl/
       fnctn-decl/
       strc-decl/
       bind-decl/
       macro-decl/
       alloc-decl/
       map-decl/
       label-decl/
       un-decl/
       ext-fwd-decl/
       swtch-decl/
       reqre-decl/
```

Before proceeding with a detailed discussion of the declarations, we shall give an intuitive overview of the effect of these declarations.

1.19.1 Scope and the Concept of GLOBAL in a Block Structured Language

As in ALGOL, the concept of GLOBAL scope conflicts with BLISS-10 block structure. This section attempts to delineate the lines of conflict.

The following types of identifiers will be classed under the general heading GLOBAL, and the word GLOBAL as it appears below will refer to this class unless specifically noted otherwise. An identifier is considered GLOBAL if it is declared in any of the following types of declarations:

```
GLOBAL
GLOBAL ROUTINE
GLOBAL BIND
GLOBALLY INDEXES
GLOBALLY NAMES
```

Whereas the BLISS-10 compiler distinguishes identifiers by using up to the first 10 characters, limitations in the current DECsystem-10 loader prevent the loader from handling names which are over six characters in length. As a result beginning with version 3 of BLISS-10, the compiler will flag the declaration of any GLOBAL name if that GLOBAL name is identical in the first six characters to any other previously declared GLOBAL name.

Bear in mind that the BLISS-10 compiler is a single pass compiler. As a result, the scope of a GLOBAL name extends from the point at which it is declared as a GLOBAL through the end of the module, regardless of the block in which it is declared.

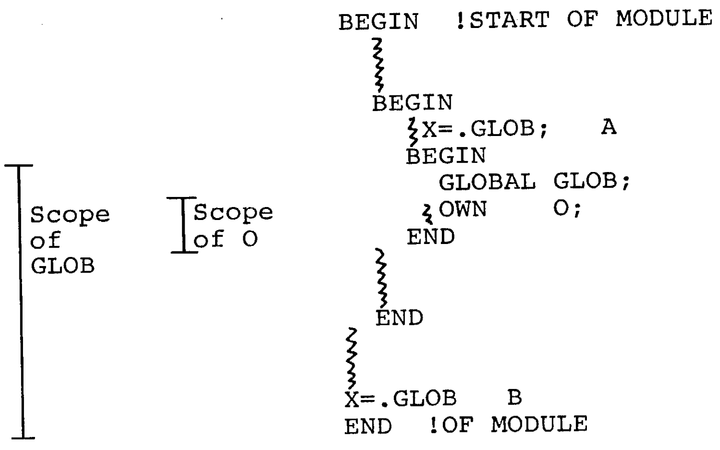
Ideally we would like to say that a name declared GLOBAL is treated as if it were declared in an imaginary block external to the outermost block of the module. This ideal could be realized if the BLISS-10 compiler were not a single pass compiler.

In any case, a name declared GLOBAL is available to the loader in order to resolve external requests at load time. GLOBAL names and routine names are the only non-system names within a BLISS-10 source program that generate symbol table entries.

A name declared as a GLOBAL may be redeclared at lower block levels as a non-GLOBAL in accordance with standard scope rules.

Examples follow in which all BLISS-10 source is omitted except for BEGIN-END pairs and certain illustrative source lines.

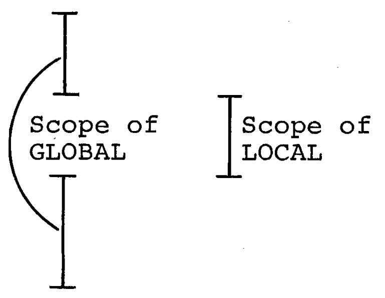
1.



```
BEGIN  !START OF MODULE

  BEGIN
     X=.GLOB;    A
     BEGIN
        GLOBAL GLOB;
        OWN    O;
     END

  END

X=.GLOB   B
END  !OF MODULE
```

Note that in example 1 the scope of the GLOBAL name extends beyond the block in which it is declared; whereas the OWN name obeys the standard scope rules. At point (A) the reference to GLOB would result in the generation of a warning message since GLOB has not yet been declared. At (B) GLOB is declared.

2.



```
BEGIN
  BEGIN
    GLOBAL GLOB;
    ≀
    BEGIN
      LOCAL GLOB
      ≀
    END
  ≀
  END
≀
END
```

In example 2, within the standard scope of the LOCAL declaration, all references to GLOB will be to the LOCAL. Outside this scope references to GLOB refer to the GLOBAL. Note again that the scope of the GLOBAL extends beyond the block in which it was declared.

1.20 STORAGE (AN INTRODUCTION)

A BLISS-10 program operates with and on a number of storage "segments". A storage segment consists of a fixed and finite number of words, each of which is composed of a fixed and finite number of bits (36).

In practice a segment generally contains either program or data; if the latter, it is generally integer numbers, floating point numbers, characters, or pointers to other data. To a BLISS-10 program, however, a segment merely contains a pattern of bits.

Segments are introduced into a BLISS-10 program by declarations, called allocation declarations (alloc-decl), For example:

```
GLOBAL G;
OWN X,Y[5],Z;
LOCAL P[100];
REGISTER R1,R2;
ROUTINE F(A,B) = .A↑.B;
```

Each of these declarations introduces one or more segments and binds the identifiers mentioned (e.g., G, X, Y, etc.) to the name of the first byte of the associated segment. (The ROUTINE declaration (routn-decl) also initializes the segment named "F" to the appropriate machine code.)

The segments introduced by these declarations contain one or more words, where the size may be specified (as in LOCAL P[100]), or defaulted to one (as in GLOBAL G;). The identifiers introduced by a declaration are lexically local to the block in which the declaration is made, with one exception - namely, GLOBAL identifiers are made available to other, separately compiled modules. Segments created by OWN, GLOBAL, and ROUTINE declarations are created only once and are preserved for the duration of the execution of a program. Segments created by LOCAL and REGISTER declarations are created at the time of block entry and are preserved only for the duration of the execution of that block. REGISTER segments differ from LOCAL segments only in that they are allocated from the machine's array of 16 general purpose (fast) registers. Re-entry of a block before it is exited (by recursive routine calls, for example) results in the dynamic allocation of LOCAL and REGISTER segments on each incarnation (reentry) of the block.

There are two additional declarations whose effect is to bind identifiers to names, but which do not create segments; examples are:

```
          EXTERNAL        S;
          BIND    Y2 = Y+2, PA = P+.A;
```

An EXTERNAL declaration binds one or more identifiers to the names represented by the same identifier declared GLOBAL in another, separately compiled module. The BIND declaration binds one or more identifiers to the value of an expression at block entry time. At least potentially, the value of this expression may not be calculable until run time - as in 'PA = P+.A' above.

1.21 INTERLUDE 2

We have reached the point where we must discuss one of the most important BLISS-10 concepts--that of structures. Understanding the structure mechanism directly involves the complete declaration syntax and indirectly the understanding of BLISS-10 routines. Our exposition will proceed as follows

1. Discuss BLISS-10 routines and functions;

2. Discuss structures, omitting complete syntactical details;

3. Discuss complete declaration syntax.

1.22 FUNCTIONS AND ROUTINES

```
          Fctn-decl::=FUNCTION name(name-lst)=E/
                      FUNCTION name=E/
                      ROUTINE name(name-lst)=E/
                      ROUTINE name=E/
                      GLOBAL ROUTINE name(name-lst)=E/
                      GLOBAL ROUTINE name=E
```

The FUNCTION declaration (fctn-decl) defines the name to be that of a potentially recursive and re-entrant function whose value is the expression E. The syntax of a normal function call is

```
       fctn-call          ::=fctn-expr(exp-lst)/fctn-expr()
       fctn-expr          ::=literal/name/cmpnd-expr/block/
                             name[expr-list]
       expr-lst           ::=E/ expr-lst,E
```

where fctn-expr is a primary expression, and must evaluate, either at compile time or runtime, to a name which has been declared as a FUNCTION. The names in the name-lst of the declaration define (lexically local) names of formal parameters whose actual values on each incarnation are determined by the expr-lst at the call site. All parameters are implicitly call-by-value; but notice that call-by-reference is achieved by simply presenting addresses at the call site. Parentheses are required at the call site even for a routine or function with no formal parameters since the name on its own is simply a pointer to the ROUTINE and not a request for an invocation. Extra actual parameters above the number mentioned in the namelist of the FUNCTION (or ROUTINE) declaration are always allowed; however, too few actual parameters can cause erroneous results at run time.(1) A ROUTINE differs from a FUNCTION in having an abbreviated and hence faster prolog. Restriction: a ROUTINE may not refer directly to local variables declared outside it, nor may it call a FUNCTION.

To eliminate possible misunderstanding on the BLISS-10 parameter passing mechanism for those readers familiar with other higher level languages, and to make the process explicit for others, we need to examine it in more detail.

BLISS-10 FUNCTIONS have one memory location set aside for each formal parameter declared (allocated dynamically on the stack). When a call on the function occurs, an evaluation of the actual parameters takes place, and the value resulting from expression evaluation becomes the contents of the location in the routine established for the corresponding formal. Because every expression in BLISS-10 computes a value, and because the value thus computed in a routine call establishes the contents of the formal parameter, we say that all BLISS-10 calls occur as calls-by-value.

Examples:

```
<1>  !The routine ARITHSUM calculates the arithmetic
     !sum of the first n integers.
     ROUTINE ARITHSUM(N)=(.N*(.N+1))/2

     Call:
          Y=ARITHSUM(10)
```

The call mechanism evaluates the actual parameter and deposits it in the location, N, in the called routine; then invokes the routine. The routine then operates on the passed value. Note the formal, N, has the dot operator applied.

```
     Call:
          Y=ARITHSUM(A)   !A names the location of the
                          !argument
```

----

(1)Note: If extra parameters are presented, and say, n are expected, then the rightmost n actual will correspond to the formal parameters; if too few actual parameters are presented they will correspond to the rightmost n formals. See Chapter 4 for details of the access mechanism.

In this case A, as a value, becomes the address of A and the call mechanism deposits this value into the location reserved for the formal, N. The value returned then equals the arithmetic sum of the address treated as an integer--hardly the outcome intended. To achieve the desired result requires the call:

```
     Y=ARITHSUM(.A)
```

This deposits the contents of A into the location of the formal, N.

The above example amplifies the strict call-by-value parameter passing in BLISS-10. Specifically, we have a routine with a list of formal names and for each formal name a location. At the call site, each actual parameter (any expression except a block) is evaluated and the contents of the corresponding formal receives the value. The caller, using the requirements of the routines, sets up the call to effect the proper initialization of the formals.

<2> The occurrence of a primary expression followed by a parameter list enclosed in parentheses triggers a function call; an identifier by itself, even one declared as the name of a routine merely denotes an address.

The value of a routine call results from the execution of the body of the routine. Consider:

```
     BEGIN
       GLOBAL X,Y,Z;
        ROUTINE F(A,B)=(.A=.B);
       Y=5;
       Z=F;
       (.Z)(X,.Y);
       .X
     END
```

This block has as its value, 5.

Z=F stores the address of F into Z but does not trigger a call. The expression

```
     (.Z)(X,.Y);
```

is actually a routine call which results in storing the contents of Y into X. It's worth emphasizing that a routine call need not explicitly name a routine by its associated identifier, only that the primary expression evaluate to a routine segment which can properly accept the arguments.

1.22.1 GLOBAL

A routine name is like an OWN name in that its scope is limited to the block in which it is declared and its value is already initialized at block entry. The prefix GLOBAL changes the scope of the routine to that of the outer block of the program enveloping all the modules. Note that this inhibits a GLOBAL routine from access to register names declared outside it.

1.22.2 EXTERNAL and FORWARD Declarations

```
ext-fwd-decl::=EXTERNAL name-par-lst/
               FORWARD name-par-lst

name-par-lst::= name-par / name-par-lst, name-par

name-par::= name(E) / name
```

EXTERNAL and FORWARD each tell the compiler how many parameters, given by E(1), are expected by an undeclared routine name. FORWARD is for routines declared later in the current block; and EXTERNAL is for routines from another module. The compiler permits the number of actual parameters in a routine call to be greater than, equal to, or less than the number of formals declared. This argument may be entirely omitted.

1.23 DATA STRUCTURES (AN INTRODUCTION)

Two principles were followed in the design of the data structure facility of BLISS-10:

1. The user must be able to specify the accessing algorithm for the elements of a structure.
2. The representational specification and the specification of algorithms which operate on the information represented must be separated in such a way that either can be be modified without affecting the other.

The definition of a class of structures (i.e., of an accessing algorithm to be associated with certain specific data structures) may be made by a declaration of somewhat the following form:

```
STRUCTURE name[strc-frml-lst]=E
```

----

(1) Clearly E must evaluate to a constant at compile time.

Particular names may then be associated with a structure class (i.e., with an accessing algorithm) by another declaration of somewhat the same form:

```
MAP strc-name name-lst
```

consider the following example:

```
<1>  BEGIN
       STRUCTURE ARY2[I,J]=
          .ARY2+(.I-1)*10+(.J-1)
       OWN X[100],Y[100],Z[100];
       MAP ARY2 X:Y:Z;
       .
       .
       .
       X[.A,.B] = .Y[.B,.A];
       .
       .
       .
     END;
```

In this example we introduce a very simple structure, ARY2, for two dimensional (10x10) arrays, declare three segments with names X, Y, and Z bound to them, and associate the structure class ARY2 with these names. The syntactic forms X[E1,E2] and Y[E3,E4] are valid within this block and denote evaluation of the accessing algorithm defined by the ARY2-structure declaration (with an appropriate substitution of actual for formal parameters).

The access algorithm on the right of the equal sign in a STRUCTURE declaration is just another expression and it computes a value.

Although they are not implemented in this way, for purposes of exposition one may think of the STRUCTURE declaration as defining a routine with one more formal parameter than is explicitly mentioned. For example, the STRUCTURE declaration in the previous example.

```
STRUCTURE ARY2[I,J] = (.ARY2+(.I-1)*10+(.J-1));
```

conceptually is identical to the ROUTINE declaration

```
ROUTINE ARY2(F0,F1,F2) = (.F0+(.F1-1)*10+(.F2-1));
```

The expressions X[.A,.B] and Y[.B,.A] correspond to calls on this routine - i.e., to ARY2(X,.A,.B) and ARY2(Y.,B,.A).

Since in a STRUCTURE declaration there is an implicit, unnamed formal parameter, the name of the structure class itself is used to denote this zero-th parameter. This convention maintains the positional correspondence of actuals and formals. Thus, in the example above, .ARY2 denotes the value of the name of the particular segment being referenced, and X[.A,.B] is equivalent to:

```
(X+(.A-1)*10+(.B-1))
```

The value of this expression is a pointer to the designated element of the segment named by X. Remember, routines have locations assigned to each of their formals. To continue the analogy three locations ARY2, I, J, would exist, and on call invocation, ARY2 would contain the value of X, I the value of .A, and J the value of .B as shown in figure 1-14. Readers should view the above example relating STRUCTURES and ROUTINES as a pedagogical aid and not as a literal description of an implementation since valid STRUCTURE accesses can be constructed where the analogy does not hold.

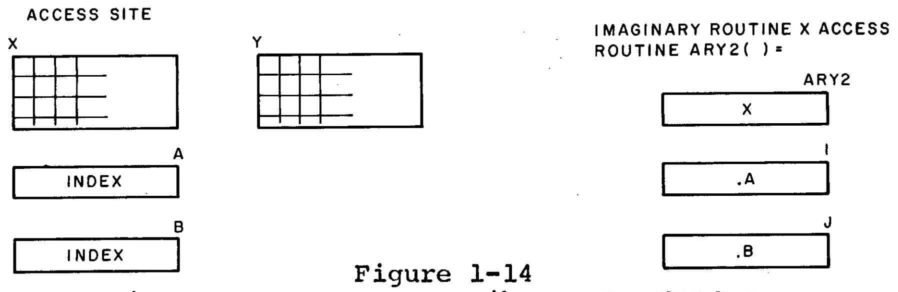


Figure 1-14
Imaginary ROUTINE to Describe a STRUCTURE Access

In the following example the STRUCTURE facility and BIND declaration have been used to encode a matrix product (Z(I,J) = X*Y (vector multiplication)). In the inner block the names XR and YC are bound to pointers to the base of a specified row of X and column of Y respectively. These identifiers are then associated with structure classes which allow one-dimensional access.

```
BEGIN
  STRUCTURE     ARY2[I,J]=(.ARY2+(.I-1)*10+(.J-1)),
  STRUCTURE     ROW[I] = (.ROW+(.I-1)),
  STRUCTURE     COL[J] = (.COL+(.J-1)*10);
  OWN X[100],Y[100],Z[100];
  MAP ARY2 X:Y:Z;
  .
  .
  .
  INCR I FROM 1 TO 10 DO !Starts a new row
      BEGIN BIND XR=X[.I,1]], ZR=Z[.I,1];
            MAP ROW XR:ZR;
       INCR J FROM 1 TO 10 DO !Steps to next column
        BEGIN
         REGISTER T; BIND YC=Y[1,.J]; MAP COL YC;
         T = 0;
         !Perform vector multiply to establish
         !one element of Z
         INCR K FROM 1 TO 10 DO T = .T+.XR[.K]*.YC[.K];
         ZR[.J] = .T;
        END;
      END;
  .
  .
END
```

Notes:
    The XR BIND establishes a row pointer in X
    The YR BIND establishes a column pointer in Y
    The ZR BIND establishes a row pointer in Z
    T accumulates the vector multiply for a given row and column
    ZR[.J] establishes a unique element in Z; ZR via the BIND selects the row and .J the element within the row. Figure 1-15 shows memory at the beginning of the calculation for Z(2,3)

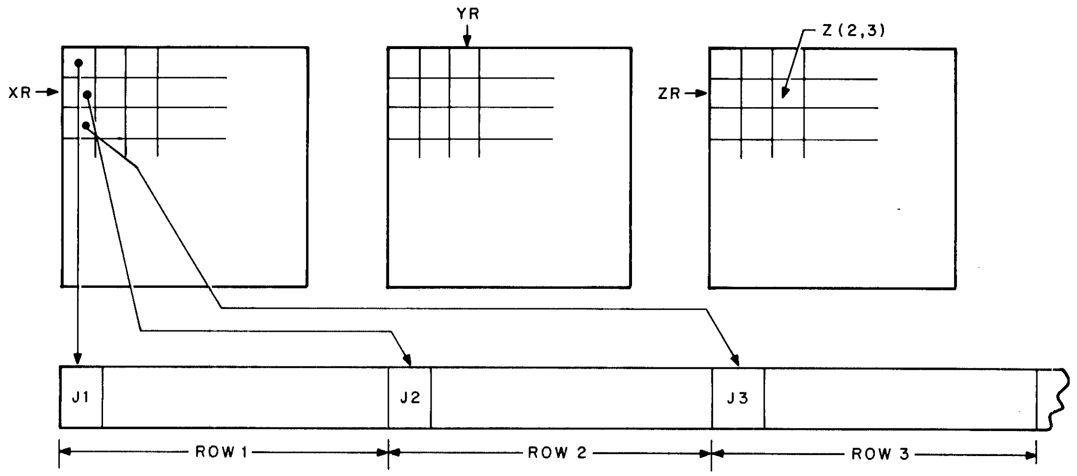


Figure 1-15
Memory at the Beginning of Calculation z(2,3)

1.24 THE ACTUAL DECLARATION SYNTAX

The example declarations in the preceding two sub-sections are valid BLISS-10 syntax; however, they do not reflect the complete power of the declarative facilities. The following sections (1.24-1.31) are definitive presentations of the actual syntax and semantics of these declarations. The actual declarations presented in the following sections differ from the examples given previously in that they admit greater interaction between the allocation declarations and structure declarations. Indeed, allocation declarations and structure declarations always interact either explicitly or by default. For example, the declaration:

```
OWN  X[100]
```

interacts with a default structure of the following form:

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

The allocation declaration, OWN X[100], will result in the allocation of 100 words permanently allocated to the block in which it appears. Let's examine how this simple declaration interacts with the default structure declaration. To do this we need to examine the detailed declaration syntax for structures.

1.24.1 Structures

```
        strc-decl::=STRUCTURE name strc-fml-lst=strc-size E
        strc-fml-lst::=[name-lst]/empty
        strc-size::=[E]/empty
```

In the structure of 1.24 we can make the following equivalences:

```
          name==VECTOR
          strc-fml-lst==[I]
          strc-size==[I]
          E==(.VECTOR+.I)<0,36>
```

Given these components of a structure declaration, how does the interaction between structure and allocation declarations occur?

Structure declarations serve to define a class of data structures by defining an explicit access algorithm, E, to be used in accessing elements of that structure. The class of structures introduced by such a declaration is given a name which may be used as the structure name in an allocation declaration or MAP declaration.

The accessing algorithm may contain declarations or blocks. However, the occurrence of a block will force compilation of the structure as if it were a ROUTINE.

The names in the structure formal list (strc-fml-lst) are formal parameter identifiers which are used in two distinct ways:

1. Dotted occurrences of the formal names positionally correlate with the values of expr-lst elements at the site of a structure access. (Recall that a structure access is syntactically <name>[expr-lst].) These are referred to as access formals and access actuals respectively.

2. Undotted occurrences of the formal names positionally correlate with the values of the expr-lst elements at the site of the declaration which associated the variable name with the structure class. These are referred to as incarnation formals and incarnation actuals respectively.

In addition to the explicit formal names, the structure name in dotted form is used as an access formal to denote the name of the specific segment being accessed (that is, to denote the pointer to the base of the segment).

In the declaration OWN X[100]

```
          X==name
          100==expr-lst
```

and the interaction between the default structure, VECTOR, and X as follows:

The incarnation actual, 100, replaces all undotted occurrences of the incarnation formal, I, which in turn determines the segment size requirement for the structure.

An access of the structure, like

```
        Y=X[10]
```

causes replacement of dotted occurrences of the formal parameters (access formals) with access actuals, in this case 10. The name itself, namely X, replaces the name VECTOR to provide a base pointer to the structure.

The default structure may be defined by the STRUCTURE declaration. Defining a structure with the name "VECTOR" causes that structure to become the default for the scope of the name. Note also that if the compiler is stopped by the ↑C↑C procedure and then STARTed, the default VECTOR structure is not redefined. In other words, if a program redefines the VECTOR structure, and the user ↑C↑C STARTs while VECTOR has been redefined, it will retain the user defined structure until after redefined explicity within the program, or re-initialized by the "R" or "RUN" monitor commands. In this instance, the ↑C↑C START procedure is not identical to the "R BLIS10" sequence.

This example shows the interaction of a simple allocation declaration with a syntactically complete structure declaration; it remains now to discuss the complete syntax of allocation declarations.

Limitation:

The compiler will not accept more than 15 formal parameters in a structure declaration. The appearance of more than 15 parameters will result in a fatal syntax error.

1.25 MEMORY ALLOCATION

To simplify our description we present two forms of the allocation declaration. The syntax used to describe statements in other DEC manuals (like PDP-11 DOS) and BNF.

Allocation Declaration

```
          OWN
          LOCAL
          GLOBAL    alloc-decl-arg-list
          EXTERNAL
          REGISTER
```

where

```
alloc-decl-arg-lst::=name:name...[expr,expr...]/
  alloc-decl-arg-lst,name:name...[expr,expr...]

struc-name::=name/empty
```

BNF Description

```
alloc-decl::=alloc-typ msid-lst
alloc-typ::=OWN/LOCAL/GLOBAL/EXTERNAL/REGISTER
msid-lst::=msid-elmt/msid-elmt,msid-lst
msid-elmt::=strc sized-chnks
strc::=strc-name/empty
size-chnks::=size-chnk/size-chnk:size-chnks
size- chnk::=id-chnk/id-chnk[expr-lst]
id-chnk::=name/name:id-chnk
```

As with most other declarations, the allocation declarations introduce names whose scope is the block in which the declarations occur. REGISTER and LOCAL declarations cause allocation of storage at each block entry (including recursive and quasi-parallel ones), and corresponding de-allocation on block exit. Storage for OWN and GLOBAL declarations is made once (before execution begins) and remains allocated during the entire execution of the program. EXTERNAL declarations do not allocate storage, but cause a linkage to be established to storage declared with the same name in a GLOBAL declaration of another module. Space for allocation is taken from core for LOCAL, OWN, and GLOBAL declarations, and from the machine's high speed registers for REGISTER declarations.

The initial contents of allocated memory is not defined and should not be presumed.

Each msid-elmt defines a set of identifiers and simultaneously maps these identifiers onto a specified structure. (If the structure part is empty, the default structure VECTOR is assumed.) Each sized chunk (sized-chnk) allows, by interaction with the associated structure of the msid-elmt, specification of the size of the segment to be allocated, and the values of the undotted structure formal to be used in accessing an instance of the structure.

If present, the structure size, i.e., [E], is used to calculate (from the incarnation actuals) the size of the segment to be allocated by an allocation declaration. After substitution of incarnation actuals, this expression must evaluate to a constant at compile time. If the structure size is omitted, it defaults to the product of the structure formals.

The example <1> in section 1.23 of a two-dimensional array might now be written:

```
<1>  BEGIN
       STRUCTURE ARY2[I,J]=
           =[I*J](.ARY2+(.I-1)*J+(.J-1));
       OWN ARY2 X:Y:Z[10,10];
       .
       .
       .
       X[.A,.B] = .Y[.B,.A];
       .
       .
       .
     END;
```

Here the mapping occurs in the OWN declaration and the segment size determination occurs by evaluating the access formals [I*J] after substituting the access actuals [10,10] and performing the size computation. In addition to the general syntax described above, for REGISTER declarations only, the following is permitted:

```
REGISTER spec-reg-lst
```

where

```
spec-reg-lst::=spec-reg/spec-reg-lst,spec-reg
spec-reg::=struc-name size=E
size::=[expr-lst]/empty
```

The expression, E, must evaluate to a constant at compile time such that 0 LEQ E LEQ 15 and the E(th) register has been reserved in the module head (see section 3.5). Thus, for example, the following declaration is legal:

```
REGISTER R=5,RX=10;
```

and will cause the specific registers 5 and 10 to be given names 'R' and 'RX', respectively. This mechanism may, for example, be used for global register communication between modules.

More Examples:

<2> LOCAL A,B,C;

Dynamically allocates 3 locals on the stack and binds the names A,B, and C to them.

<3> OWN A:B:C[3]

Allocates 9 words, binds the names A, B, and C to the first, fourth and seventh word, respectively and maps the default "VECTOR" structure onto them.

```
<4> STRUCTURE BITVECTOR[I]=[I↑(-5)+1]...;
    GLOBAL    BITVECTOR A:B[300]:C[200],X[20];
```

Allocates 4 chunks of global storage with sizes (300/32+1), (300/32+1), (200/32+1) and 20, respectively, binds the names A, B, C and X to them, and maps BITVECTOR onto A, B, and C and then default VECTOR structure onto X, making the names A, B, C and X globally available to other loaded modules.

1.26 MAP DECLARATION

```
map-decl::=MAP msid-lst/MAP link-name msid-lst
```

The MAP declaration is syntactically and semantically similar to an allocation declaration except that no new storage or identifiers are introduced. The purpose of the MAP declaration is to permit redefinition of the structure and expr-lst information associated with an identifier (or set of identifiers) for the scope of the block in which the MAP declaration occurs.

Examples

<1> MAP A:B[10];

Maps the default structure VECTOR onto A and B, associating 10 with the first incarnation formal.

<2> MAP A;

Maps the default structure VECTOR onto A, associating the default value 1 with the first incarnation formal.

<3>
```
MAP SOMESTRUCT A:B:C,
    ANOTHERSTRUCT X[3]:Y:Z[5,4]
```

SOMERSTRUCT is mapped onto A, B, and C, with default incarnation actuals of 1; ANOTHERSTRUCT is mapped onto X, Y and Z. Incarnation actuals 3 and (default) 1 are associated with X; 5 and 4 with both Y and Z.

NOTE

In the above, A, B, C, X and Y must have been declared previously.

1.27 BIND DECLARATIONS

```
bind-decl::=BIND equ-lst/GLOBAL BIND equ-lst
equ-lst::=equ/equ, equ-lst
equ::=msid-elmt=E
```

A BIND declaration introduces a new set of names whose scope is the block in which the BIND declaration occurs, and binds the value of these names to the value of the associated expressions at the time that the block is entered. Note that these expressions need not evaluate at compile time.

Syntax for GLOBAL BIND is the same as that for the ordinary BIND declaration with two exceptions. The word GLOBAL precedes the word BIND and all expressions to which names are bound must be constants at compile time.

If an expression to be globally bound is not a compile time constant, a warning message will be generated and the bind will be treated as an ordinary non-global bind. The effect of the GLOBAL BIND is to generate a symbol with an associated 36 bit value which is available to separately compiled modules.

The user is warned that although BLISS-10 generates 36 bit values for GLOBAL BINDs, current loader limitations may result in the use of only the right half of the value at load time.

Example

```
<1>  BIND ONE=1,CTWO=.2<0,36>,NAME=ALOCAL,
          Y[5]=QQ[20],R:L:M=7;
```

Future references to:

"ONE" will be equivalent to using "1";
"R", "L", and "M" to using 7;
"NAME" to using "ALOCAL";
"CTWO" to the contents of register 2 at the time the bind is executed;
"Y[E]" to "QQ[20+E]" (assuming QQ was also mapped with the vector structure).

NOTE

The Y[5] only indicates that the default vector structure should be mapped with incarnation actual of 5 - not that of (Y+5)=QQ[20].

1.28 LABEL DECLARATION

```
labl-decl::=LABEL
labl-lst labl-lst::=name/name, labl-lst
```

Labels are used solely in conjunction with the LEAVE expressions. Each name to be used as a label must be so declared at the head of the block containing that usage. Any specific label may be used only once within the lexical scope of its definition.

1.29 REQUIRE DECLARATION

```
reqre-decl      ::=REQUIRE file-spec
file-spec       ::=file-name-spec/device:file-name-spec
device          ::=name
file-name-spec  ::=file-name/file-name[ppn-spec]
file-name       ::=name/name.name
ppn-spec        ::=DECppn
DECppn          ::=octal,octal
```

The REQUIRE declaration instructs the compiler to read the text of the file named by the <file-spec> as if that text had been copied into the source file immediately following the semicolon which follows the declaration. A REQUIREd file may in turn REQUIRE another file to a total depth of six REQUIREd files.

A word about the defaults for omitted parts of a <file-spec>: <device> is defaulted to DSK, [<ppn-spec>] is defaulted to the project-programmer number of the job which is running the compiler, and omitted parts of a <filename> are defaulted to blank.

The following restrictions should be noted:

1. Everything between the semicolon which follows a REQUIRE declaration and the next carriage return will be ignored.

2. Only the first six (six, three) characters of the <name> given for a device (file, extension) will be used. These names should refer to existing devices and files.

3. The <ppnspec> should refer to a valid account number on the system being used.

4. Recursive use of REQUIRE is illegal as a result of the depth restriction.

1.30 MACROS

In order to facilitate program readibility and modifiability, a macro system is embedded in BLISS-10. The system allows nested macro definition as well as iterative and recursive forms of evaluation.

1.30.1 Syntax for Macro Declarations

```
mac-decl::=MACRO dfn-lst
dfn-lst::=dfn/dfn-lst,dfn
dfn::=name(name-lst)=strng-no-$ $/
      name=strng-no-$ $
```

The essential function of the macro system is to replace the macro name and its actual-parameter list (wherever the name occurs within the scope of the macro definition in the program) by its body, with actual-parameters substituted for formals. The body is considered to be a string of "atoms"--names, literals and delimiters--and is therefore independent of editing symbols--blanks, CR,LF, and BLISS-10 comments--once the atoms are determined at macro declaration time. Atoms are the smallest recognizable syntactic elements in the BLISS-10 language.

The format of the macro call is simply the macro-name followed by the bracketed actual-parameter list. The brackets must be the pair: (); and the actual-parameters must be separated by commas. The actual parameters themselves may be arbitrary strings of atoms; however, occurrences of the brackets: (), [], <>, and components of a parameter must be properly balanced and nested. All macros in actual-parameter lists are expanded before formal/actual binding; we can say BLISS-10 expands macros from the inside out.

A balanced string is any string for which the number of right brackets (",", "]", or ">") in the string equals or exceeds the number of corresponding left brackets. This includes the null string. A balanced string is associated with the formal parameter in the corresponding ordinal position in the definition. Caution: "(" is not a balanced string; there is no matching close parenthesis.

Macros effect a text substitution process, involving a definition and a call. The actual parameters in the call serve as the 'working text' for formal parameters in the definition and, on call, the definition serves as a template for a text substitution process resulting in a macro expansion.

Note that

1. "extra" balanced strings will be simply ignored, but parsed as described above.

2. Null balanced strings are accepted.

3. The macro call may present fewer balanced strings than the definition, in which case the null string will be used for the "missing" arguments.

4. A call must have a balanced string list if the definition had a namelist.

The expanded string from a macro replaces the macro call in the program prior to lexical processing, and scanning resumes at the head of this string. Hence macro calls may be nested. Indeed, parts of a "nested" call may come from the actual parameter(s) of the containing macro, from the body of the containing macro or even from the text following the containing macro.

As with other declarations, macros have a scope given by the block in which they are defined - with this exception: any macro being expanded at the end of a block will, in effect, be purged, but its expansion will run to completion. This might occur, for example, if a macro contained an END as in:

```
BEGIN
   MACRO QQSV=END B←"TQ" $;
   QQSV
END
```

This may lead to anomalous behavior depending on the specific program.

EXAMPLES:

Macros may be used to provide names to bit fields so as to improve readability.

```
MACRO EXPONENT = 27,8 $;
MACRO MANTISSA = 0,27 $;
MACRO SIGN = 35,1 $;
LOCAL X;
X<SIGN>←0; X<EXPONENT>←27; X<MANTISSA>←.I;
```

Macros may be used to extend the syntax in a limited way.

```
MACRO NEG = 0 GTR $;
MACRO UNLESS(X)=IF NOT (X) $;
```

Macros may be used to effect in-line coding of a function.

```
MACRO ABSOLUTE(X)=BEGIN REGISTER TEMP;
            IF NEG(TEMP←X) THEN -.TEMP ELSE .TEMP END $;
! HERE THE ACTUAL PARAMETER SUBSTITUTED FROM X MAY NOT
! INCLUDE THE NAME TEMP.
```

1.31 UNDECLARE DECLARATION

```
un-decl::=UNDECLARE name-lst
```

The identifiers in the name-lst become undefined within the scope of the declaration.

CHAPTER 2
ADDITIONAL LANGUAGE FEATURES

2.0 SPECIAL LANGUAGE FEATURES

The previous chapter described the basic features of the BLISS-10 language. In this chapter we describe additional features which are highly machine and implementation dependent.

2.1 SPECIAL FUNCTIONS

A number of features have been added to the basic BLISS-10 language which allow greater access to the PDP-10 hardware features. These features have the syntactic form of function calls and are thus referred to as "Special Functions". Code for Special Functions is always generated in-line. The names of all special functions are reserved words.

2.2 CHARACTER MANIPULATION FUNCTIONS

Nine functions have been specified to facilitate character (any list of ordered bytes) manipulation operations. They are:

```
SCANN (AP)                  COPYNN (AP1, AP2)

SCANI (AP)                  COPYNI (AP1, AP2)

REPLACEN (AP, E)            COPYIN (AP1, AP2)
REPLACEI (AP, E)            COPYII (AP1, AP2)

                  INCP (AP)
```

For each of these E is an arbitrary expression, and AP is an expression whose value is a pointer to a byte pointer. The second of these byte pointers is assumed to point to a character in a string.

SCANN (AP)	is a function whose value is the character ..AP.
SCANI (AP)	is like SCANN except that, as a side effect, the string pointer is set to point at the next character of the string before the character is scanned. I.e. is equivalent to (INCP(AP);SCANN(AP))
REPLACEN (AP, E)	is a function whose value is E and which as a side effect, replaces the string character by E. Equivalent to .AP=E.
REPLACEI (AP, E)	is similar to REPLACEN except that the string pointer is set to point at the

	next character of the string before the value of E is stored. Equivalent to (INCP(AP);REPLACEN(AP,E))
COPYNN (AP1, AP2) COPYNI (AP1, AP2) COPYIN (AP1, AP2) COPYII (AP1, AP2)	these functions are similar in that they each effect a copy of one character from a source string (pointed at by .AP1) to a destination string (pointed at by .AP2). However, COPYNN advances neither pointer, while COPYNI advances .AP2, COPYIN advances .AP1, and COPYII advances both. In each case the pointer is advanced before the copy is effected. The value of each function is the character that is copied.
INCP (AP)	advances .AP to point to the next character in the string. This function has the value zero.

Suppose that a string (of 7-bit ASCII characters) is stored in memory beginning at location S. The string is terminated by a null (zero) character. The following skeletal code will transform it into a string of 6-bit (SIXBIT) characters with blanks deleted:

```
BEGIN
REGISTER P7, P6, C;
P7 ← (S-1) <1, 7>; P6 ← (S-1) <0,6>;
WHILE (C ← SCANI (P7)) NEQ 0 DO
     IF .C NEQ "  " THEN REPLACEI (P6, .C-#40);
...
END;
```

2.3 MORE SPECIAL FUNCTIONS

1. SIGN(E)==	-1 if E < 0 0 if E = 0 +1 if E > 0
2. ABS(E)==	E if E > 0 E if E = 0 -E if E < 0
3. FIRSTONE(E)==	-1 if E = 0 number of zero-bits to the left of the first one-bit in the value of E otherwise.
4. OFFSET(name)==	36 bit value of the stack offset of the argument (a local or formal symbol name) with respect to the stack address to which FREG is intialized upon routine or function entry. Any other type argument to the OFFSET function will generate a warning message (#422) and OFFSET will return the value 0.

2.4 MACHINE LANGUAGE

It is possible to insert DECsystem-10 machine language instructions into a BLISS-10 program in the syntactic form of a special function

```
          OP (E1, E2, E3, E4)
```

where

OP	is one of the DECsystem-10 machine language mnemonics (see table below).
E1	is an expression whose least significant 4 bits will become the accumulator (A) field of the compiled instruction. This expression must yield a value at compile time of a register name or a literal.
E2	is an expression whose least significant 18 bits will become the address (Y) field of the compiled instruction.
E3	is an expression whose least significant 4 bits will become the index (X) field of the compiled instruction.
E4	is an expression whose least significant bit will become the indirect (I) bit of the compiled instruction. E4 must evaluate to a constant at compile time.

(A table of machine language instruction mnemonics follows. Defaults for E1-E4 are 0.)

The 'value' of these machine language instructions is uniformly taken to be the contents of the register specified in the accumulator (A) field of the instruction. (This makes little sense in a few cases, but was adopted for uniformity.)

In order for the compiler to conserve space during compilation, the mnemonics for the machine language operators are not normally preloaded into the symbol table. Therefore, in order to use this feature of the language, it is necessary for the programmer to include one of the following special declarations

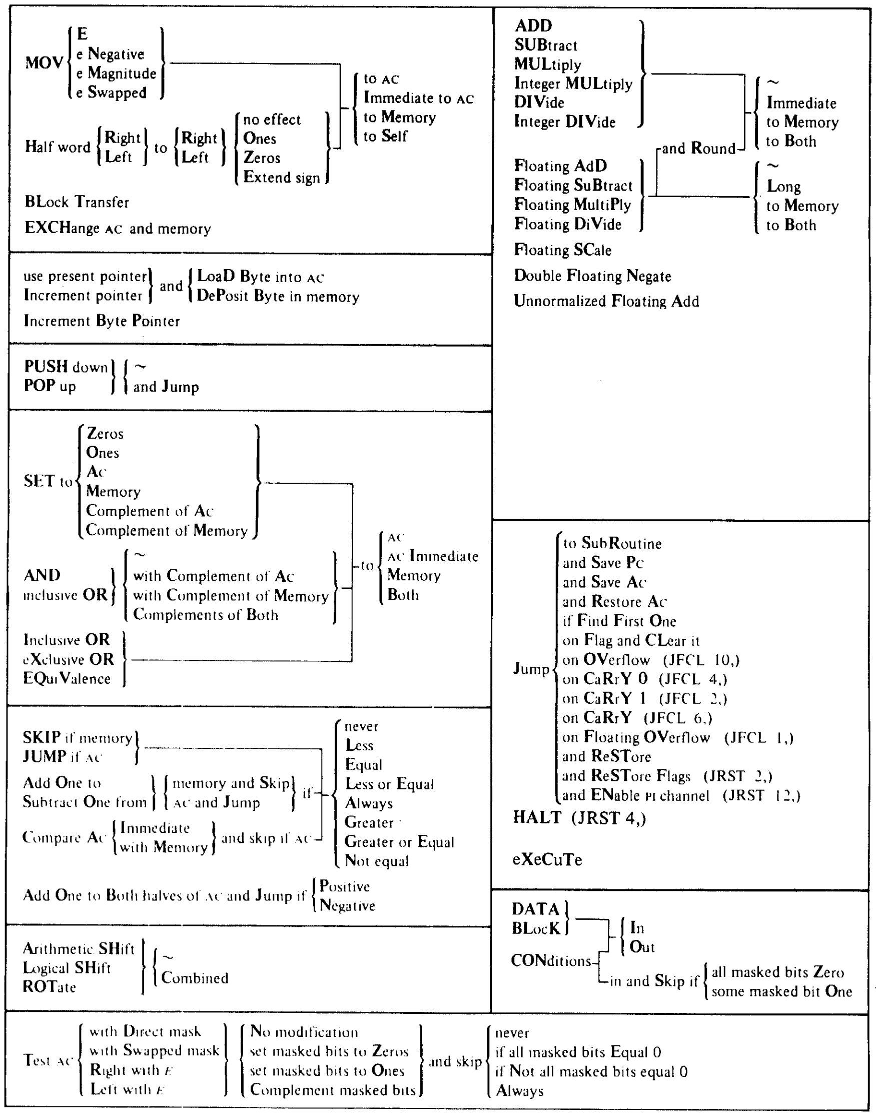
```
          mach-decl ::= MACHOP mlist / ALLMACHOP

          mlist ::= name = e / mlist , name = e
```

in the head of a block which embraces occurrences of these special functions.

NOTE

The e's in an mlist must be the high order nine bits of the actual values of the machine operation and must evaluate at compiler time.

DECsystem-10 Instruction Mnemonics


MOV E / e Negative / e Magnitude / e Swapped
to AC / Immediate to AC / to Memory / to Self
Half word Right / Left to Right / Left
no effect / Ones / Zeros / Extend sign
BLock Transfer
EXCHange AC and memory
use present pointer / Increment pointer and LoaD Byte into AC / DePosit Byte in memory
Increment Byte Pointer
PUSH down / POP up ~ / and Jump
SET to Zeros / Ones / AC / Memory / Complement of AC / Complement of Memory
AND / inclusive OR ~ / with Complement of AC / with Complement of Memory / Complements of Both
to AC / AC Immediate / Memory / Both
Inclusive OR / eXclusive OR / EQuiValence
SKIP if memory / JUMP if AC
Add One to / Subtract One from memory and Skip / AC and Jump if
Compare AC Immediate / with Memory and skip if AC
never / Less / Equal / Less or Equal / Always / Greater / Greater or Equal / Not equal
Add One to Both halves of AC and Jump if Positive / Negative
Arithmetic SHift / Logical SHift / ROTate ~ / Combined
ADD / SUBtract / MULtiply / Integer MULtiply / DIVide / Integer DIVide
~ / Immediate / to Memory / to Both
and Round
Floating AdD / Floating SuBtract / Floating MultiPly / Floating DiVide
~ / Long / to Memory / to Both
Floating SCale
Double Floating Negate
Unnormalized Floating Add
Jump to SubRoutine / and Save PC / and Save AC / and Restore AC / if Find First One / on Flag and CLear it / on OVerflow (JFCL 10,) / on CaRrY 0 (JFCL 4,) / on CaRrY 1 (JFCL 2,) / on CaRrY (JFCL 6,) / on Floating OVerflow (JFCL 1,) / and ReSTore / and ReSTore Flags (JRST 2,) / and ENable PI channel (JRST 12,)
HALT (JRST 4,)
eXeCuTe
DATA / BLocK In / Out
CONditions in and Skip if all masked bits Zero / some masked bit One
Test AC with Direct mask / with Swapped mask / Right with E / Left with E
No modification / set masked bits to Zeros / set masked bits to Ones / Complement masked bits and skip
never / if all masked bits Equal 0 / if Not all masked bits equal 0 / Always

Symbol table space for these names is released when the block in which the declaration occurs is exited. Any name declared in a MACHOP declaration and not followed immediately by an open paren will generate a fatal syntax error.

NOTE

> The description of fields E2, E3, E4 needs some simplification in the case where E2 is a name. The compiler attempts to produce a single instruction for the machine language expression whenever possible. For example, consider the expression MOVEM(5,A) where A is a local variable. The compiler, noting that the index register has been defaulted to zero, produces a 22 bit address using the F register for the index register field of the instruction.
>
> Note also that E4 must be a constant at compile time.

2.5 COMMUNICATION WITH THE MONITOR

Additional special forms may be introduced to facilitate communication with particular monitors.

CHAPTER 3
SYSTEMS FEATURES

3.1 COMPILATION CONTROL

The actions of the compiler with respect to a program may be controlled by specifications:

1. In the initial input string from a TTY,
2. In the module head, or
3. By a special switches declaration.

Not all actions can be controlled from each of these places, but many can.

Some actions once specified have a permanent effect while the effect of others can be modified (such as listing control). The table in section 3.5 gives a list of various compiler actions and the associated switch and/or source language constructs which modify those actions. This list is subject to change.

3.2 COMMAND SYNTAX

3.2.1 Normal Use

The general format of the command string is:

```
OBJDEV: FILE.EXT,LSTDEV: FILE.EXT ← SORCDEV:
          FILE.EXT,...,SORCDEV:FILE.EXT
```

The OBJDEV: FILE.EXT and/or LSTDEV: FILE.EXT may be omitted with the implication that the corresponding file is not to be generated. The .EXT may be omitted on any of the file specifications and the following defaults assumed:

object file:	.REL
listing file:	.LST
source file:	.BLI, .B10. (null extension)

The three source file defaults are all tried in the order given if no extension is explicitly typed for a source file. To specify a null file extension, a period must by typed after the filename; otherwise, a default extension will be supplied. Switches may be included in the command string as either /x or /-x (where x is a letter). A switch of the form /-x has the opposite effect of a switch of the form /x (for the same x). A switch may be included anywhere; however, some switches particularly affect the file with which they appear (and usually all those to its right).

Project-programmer numbers (PPN's) may be specified in one of two ways: if a PPN appears to the left of a file name, it applies to that file and all files to its right (unless changed by having a different PPN appear to the left of another file later); if it appears to the right of a file name, it applies only to that file. A PPN to the right of a file name always applies to that file, even if a PPN has appeared to the left of an earlier file name.

3.2.2 Use From CCL

If the COMPIL CUSP has been modified to allow BLISS-10 programs to be processed, then the COMPILE, EXECUTE, LOAD, and DEBUG commands may be used. A few words of warning, however, are necessary.

1. The BLISS-10 cross-reference switch (/C switch in the command string) is invoked by the /CREF switch, However, BLISS-10 directly generates its own cross-reference listing. Therefore, further processing with CREF is unnecessary and will produce unpredictable results.

2. The name of the REL file in the "+" construction is the name of the last file given, i.e., "A+B+C" generates files "C.LST" and "C.REL". Some BLISS-coded systems use a declaration file which forms an outer block, and an "end" file to close the outer block, with the desired programs sandwiched between, e.g.,

   ```
   "BEGIN+H1DECL+END"
   ```

   The .REL and .LST files will be END.REL and END.LST. To obtain the proper results, use the "=" construction, i.e,

   ```
   "H1DECL=BEGIN+H1DECL+END"
   ```

3.3 MODULE HEAD

As explained in 1.8 the syntax for a module is

```
module ::= MODULE name (parms) = block ELUDOM
```

The <parms> field may contain various information which will affect the compiler's action with respect to the current program. The syntax of this field is

```
parms ::= parm / parm,parms
```

The allowed forms of <parm> are given in tabular form in section 3.5 under the column headed Module Head.

3.4 SWITCHES DECLARATION

```
swtch-decl  ::= switches swtch-lst
switch list ::= swtch / swtch, swtch-lst
swtch       ::= (see entries in the "SWITCHES"
                column in section 3.5)
```

The SWITCHES declaration allows the user to set various switches which control the compiler's actions. The effect of a SWITCHES declaration is limited to the scope of the block in which the declaration is made. The allowed forms of SWITCH are given in tabular form in section 3.5 under the column headed "SWITCHES". Note that switch keywords are not necessarily reserved words.

3.5 ACTIONS

Command	Module Head	Switches	Initial Setting	Action
/L	LIST	LIST	ON	Enable listing of the source test. This switch is assumed true initially.
/K,/-L	NOLIST	NOLIST	OFF	Disable listing of the source text.
/N	NOERS	NOERS	OFF	Do not print error messages on the TTY.
/-N	ERRS	ERRS	ON	(Re)-enable error messages on TTY.
/M	MLIST	MLIST	OFF	Enable listing of the machine code generated.
/-M	NOMLIST	NOMLIST	ON	Suppress listing of generated machine code.
/H	HISEG	-	OFF	Force entire compilation into high segment. Initially modules are assumed to be two segments. (see 3.7).
/I	INSPECT	INSPECT	OFF	When true, this switch will cause a special word to be emitted immediately prior to each function or routine body. This word contains information to facilitate a SIMULA-like inspection mechanism.
/-I	NOINSPECT	NOINSPECT	ON	This sets the inspection switch false.
/X	SYNTAX	-	OFF	Syntax check only! No code will be generated - this speeds the compilation process and is therefore useful during the initial stages of program development. '/-X' is illegal.
-	DREGS=E	-	5	'E' specifies the number of 'declared'-type registers to be used.

-	RESERVE(E1,...En)	-	none	Registers with absolute names E1,...,En are reserved (usually for inter-module communication).
/O	OPTIMIZE	OPTIMIZE	ON	Because of the possibility of computed addresses in BLISS-10 programs, it is not possible for the compiler to determine whether optimization of sub-expressions is possible across ";"'s in a compound expression. Therefore the compiler operates in two modes - one in which it does optimize such common sub-expressions and one in which it does not. When the 'optimize' switch is true the compiler attempts to optimize across a ";".
/U or /-O	NOOPTIMIZE	NOOPTIMIZE	OFF	Sets the optimization switch (see above) to false.
/E	EXPAND	EXPAND	OFF	Give trace of macro expansions.
/-E	NOEXPAND	NOEXPAND	ON	Turn off trace of macro expansion.

-	SREG = E
	VREG = E
	FREG = E
		-
		-
		-
			chosen by compiler	The user may use these to choose specific registers to be used as the S, V, and F, respectively.

/C /-C	XREF NOXREF	XREF NOXREF	OFF	Print a cross-reference to all identifiers at the end of compilation (assumes a listing is being printed).
/R /-R	NORSAVE RSAVE	- -	OFF ON	The compiler normally generates code to save all declarable registers around an EXCHJ operation. This default may be overriden by a /R, or NORSAVE. RSAVE reverts to the default.

/V	LOWSEG	-	OFF	Force entire compilation into the low segment. '/-V' is illegal. (see 3.7).
-	STACK (see text at right)	-		The syntax of the module head permits automatic allocation and initialization of the run-time stack. The syntax is stack deal --> STACK/STACK= explicit-stack where explicit-stack --> stype s-name-sz stype --> GLOBAL/OWN/EXTERNAL s-name-sz --> (ss-OPTN) ss-OPTN --> / e The defaults are 'STACK'= STACK=OWN(STACK,#1000) 'STACK(e)' = STACK=OWN(STACK,e) etc.
/G	GLOROUTINES	GLOROUTINES	OFF	All routine names are forced to be 'global'.
/-G	NOGLOROUTINES	NOGLOROUTINES	ON	Non-global routines are not forced to be global.
-	ENTRIES=(nl..,nm)	-	none	An 'entry' block is created at the beginning of the '.REL' file for the names nl,n2,...nm. These names must subsequently be declared 'global' in the module. This permits FUDGE2 to be used to create a library.
/F /-F	FSAVE NOFSAVE	FSAVE NOFSAVE	OFF	Use of /F insures that FREG will be set up properly on every routine entrance and exit whether or not FREG is accessed within the routine.
-	TIMER	-	OFF	The syntax of the module head permits automatic inclusions of code to

				facilitate tracing and timing. timerdecl --> TIMER/TIMER(e)/ TIMER=explicit-timer explicit-timer --> trtype tname-size trtype --> FORWARD/EXTERNAL tname-size --> (name size-optn) size-optn --> /e The defaults are 'TIMER==TIMER = EXTERNAL (TIMER,4) 'TIMER(e)'==TIMER = EXTERNAL (TIMER,e) etc.
/T	TIMING	-	OFF	If /T appears in the command string and there is no TIMER switch in the module head, the default TIMER declaration is assumed.

				If the TIMER declaration is given in the module head, timing routine linkages will be generated.
/-T	NOTIMING	NOTIMING	ON	Turns off generation of timing linkages.
-	-	TIMING	OFF	If timing linkages were being generated and have been suppressed by the NOTIMING switch in the SWITCHES declaration, this re-enables generation. Otherwise it has no effect.
-	CCL	-	OFF	Generates a CCL-compatible entry linkage as the first two instructions of the main program. A STACK declaration must also be present to enable generation of these instructions.
/S	-	-	OFF	Results in the display on the TTY of each routine after its Macro code has been generated in the expanded listing file. If an expanded listing file has not been specified and /S is used the routine names will not be displayed. /S also prints: 1. The number of entries used in the literal table along with the size of the literal table. One word per entry.

				2. The approximate maximum number of words used in the stack since the compiler was started via a START command along with the size of the stack.
				3. The maximum compiler size during compilation in K as <low segment size> + <high segment size> K.
/P	PROLOG	-	OFF	Causes generation of PROLOG- EPILOG code in this module.
-	VERSION=<vno>	-	-	Causes generation of the specified version number in location .JBVER. <vno> is a version number in the standard DECsystem-10 format.
/A /-A	ENGLISH NOENGLISH	ENGLISH NOENGLISH	ON OFF	Prints error messages in long form (mnemonic plus message) if on, short form (mnemonic only) if off.
/B	START	START	OFF	Causes generation of a start block in the REL file if on. This switch is also enabled by the STACK declaration.
/D /-D	DEBUG NODEBUG	- -	OFF	Causes DEBUG linkage to debugging routine. The routine name of the debugging routine is assumed to be the name of the TIMER routine (specified by the TIMER module declaration
-	HEADFILE file-spec	-	-	This entry is the equivalent to a REQUIRE declaration, but within the module head. See the REQUIRE declaration for details.

3.6 UNENFORCED RESTRICTIONS

There are certain language restrictions that are not (some cannot be) enforced by the BLISS-10 compiler. Let the user take note!

1. BLISS-10 itself uses only the first ten characters of identifiers - distinct identifiers with the same initial ten characters will not be distinguished.

   In particular, the module name is used to construct certain GLOBAL symbols as explained in section 4.4 and hence should be unique from other global symbols in their first four characters. Also GLOBAL symbols must be unique in their first six characters (current LOADER/LINK restrictions).

2. The BLISS-10 compiler distinguishes between two classes of temporary registers - savable (used for declared registers, etc.) and nonsavable. The number of savable registers is determined by the DREGS command and its default declaration is DREGS=5. All BLISS-10 modules to be combined by the loader into a load-module must be compiled with identical DREGS declarations - otherwise run-time routine linkage may not work correctly.

3. Care must be exercised in invoking a FUNCTION in an improper environment. This might occur if a FUNCTION name is passed as a parameter to a ROUTINE which then invokes the FUNCTION. Displays required by the FUNCTION execution will not be set up.

3.7 SEGMENT DECLARATIONS

The HISEG and LOWSEG switches both produce one-segment programs - both pure (code, literals) and impure (local, global, own) data go into the same segment. These declarations cannot be reset once specified.

The "pure" area consists of XXXX.C, 0, F, L,., P; the "impure" area consists of XXXX.G, O. Basically, this means that all portions of core that are modified during an execution are "impure", and all sections that are fixed (constants, program code, etc.) are "pure". The following table specifies segment usage for particular switch settings.

SWITCH	PURE	IMPURE
No Switch:	high	low
HISEG:	high	high
LOWSEG:	low	low

3.8 SIX12

SIX12

SIX12 is a module containing a driver to allow interactive debugging of a BLISS program, essentially the service provided by this module is:

1. Monitor the TTY to allow the user to 'interrupt' his program and enter the debugging mode.

2. Provide analysis of the syntax of lines input to the debugger, and switching to any of several specific debug routines. A few generally useful debug routines are included in this package -- the intention, however, is that the user will build his own routines tailored to the problem and use this package merely to provide a standard interface.

The syntax of input lines is:

```
<LINE> == <EMPTY>/<ROUTINE><PARM-LIST>
<ROUTINE> == <* ANY PRINT STRING WITHOUT EMBEDDED BLANKS *>
<PARM-LIST> == <EMPTY>/<PARM>/<PARM-LIST>,<PARM>
<PARM> == <EMPTY>/<ATOM>/<PARM>+<ATOM>/<PARM>-<ATOM>
<ATOM> == <NUMBER>/<DDT-NAME>/*/$/,<ATOM>/
          <ATOM>[<PARM>]/(<PARM>)
<NUMBER> == <INPUT-BASE-NUMBER>/#<OCTAL-NUMBER>/
          #<DECIMAL-NUMBER-IF-INPUT-BASE-OCTAL>
<DDT-NAME> == <IDENTIFIER>/<IDENTIFIER>(<NUMBER>)
```

For example, the following are valid lines:

```
= X,.X,.13
← X(3),2
GLOP↑ 3,#14,.X[.Y]*+1
THUD   .(Z[.X+1-.P]-.Y),46
```

The intended interpretation of an input line is that the name debugging routine be called and the specified parameter values passed to it. The interpretation of the various atoms is:

1. Decimal and octal constants represent themselves

2. DDT-names are locked up in the DDT symbol table, the interpretation of such names is as in BLISS - that is, they are an address!

3. The symbol "*" is the value of the positionally corresponding parameter of the immed. previous debug line.

4. The symbol "$" is the address of a special "result area" -- may be used by some specific debug routines, displacements from the "result area" may be addressed for convenience as $N where N is a number.

5. The "." operator is interpreted as the "@" in BLISS.

6. The symbols "[" and "]" denote integral word displacements -- iE, E1[E2] = (E1+E2).

NOTE

If, at any time during execution, any character is typed on the TTY, the debug package will shortly intercept control and wait for the user to complete the debug line.

User-specific debugging routines may be easily added to the collection currently provided as follows:

1. Write the routine to accept its input parameters from the array "DEBUGPARMS". Note the number of parms is in the "NDEBUGPARMS".

2. Add to the PLIT, "DEBUGROUTS", a print-name and the actual routine name.

The set of debug routines provided in this version are:

= P1,...,PN	Print the values of P1,...,PN
/ P1,P2	Print the values of P1,...,P1+P2-1
← P1,P2	Assign P1←P2
<EMPTY>	No action
DDT	Call DDT
GO	Resume executing user program
BREAK P1,...,PN	Set break points at the heads of routines P1,...,PN. This is a break to this debug system (not DDT).
DBREAK P1,..,PN	Remove break points set as above.
ABREAK P1,,,PN	Set break points at the exit of routines P1,,,PN. This is a break to the debug system (not DDT). The VREG is printed.
DABREAK P1,,,PN	Remove ABREAK points set as above.
CALLS	Display the stack of routine calls
CALL+	Display call stack and locals -- the latter may not be very useful unless the user is very familiar with the BLISS run-time environment.
CALL N	Print the last N calls on stack. If N is omitted, print last call.
LCALL N	Same as above but also displays locals.
IBASE N	Sets the input number base to N (N is always decimal) (1<N<11). Note that # immediately preceeding a number always means octal input except when the default base is octal, in which case # means decimal. If N is omitted it prints the current input base in decimal.
OBASE N	Sets the output number base to N (N is always decimal) (1<N<11). If N is omitted it prints the output base in

	decimal.
WBASE N	Sets the maximum size of displacement that will be printed in 'BASE+DISPLACEMENT' output to N. (W is for Wulf for historical reasons.) initially set 1000(octal). If N is omitted, the current WBASE is printed in the current output base.
SETTRACE	Turns on trace mode. Will display call stack for routines as they are entered and left. Trace equivalent to SETTRACE + GO . EXECUTION IS RESUMED IN TRACE MODE. TRACE MODE ENDS WHEN ANYTHING IS TYPED AT THE TERMINAL OR A BREAK POINT IS REACHED.
EX P1,...,PN	Call the procedure named P1 with parameters P2,...,PN. (N<6) returns the value in "$".
EV P1,...,PN	Call the procedure named P2 with parameters P3,...,PN. (N<7) the value is placed in the location specified by P1.
PRS P1,,,PN	Prints a list of all DDT symbol table entries for P1,,,PN.
DISABLE	Turns off typein monitoring. Allows type-ahead. to resume typein monitoring, enter DDT and do PUSHJ SIXENABLE$X.
OPAQUE P1,..,PN	Makes the named routines opaque to tracing, i.e., if the routine is entered with tracing turned on, tracing is turned off until the matching routine exit is encountered. All opaque flags are reset if a break or TTY interrupt occurs.
DOPAQUE P1,.,PN	Undoes the effect of opaque, i.e. the named routines are no longer opaque.
MONITOR P1,,PN	Monitors the locations P1,,PN. If value changes, tells where and how.
DMONITOR P1,,PN	Turns off monitoring.

NOTE

In order to use this package, the modules to be debugged must be compiled with the proper compilation control, namely:

1. "TIMER=EXTERNAL(SIX12)" must appear in the module head.

2. Either "/F/T" must be used in the command string, or "FSAVE,TIMING" must appear in the module head.

Use Notes :

1. Because it is sometimes useful to enter this package immediately upon the start of execution (after stack initialization), a facility to do this has been provided. Enter DDT and type:

```
SIX1..$:     STARTFLG!  1 <CRLF>
$G
```

If your program does not use the name STARTFLG, then the SIX1..$* can be eliminated. (Note: $ is altmode)

CHAPTER 4
RUN TIME REPRESENTATION OF PROGRAMS

4.1 INTRODUCTION TO CALLING SEQUENCES

In order to make fullest possible use of BLISS-10 it is important to understand the run-time environment in which BLISS-10 programs execute. The address space is occupied by various types of information:

1. Programs

2. Constants

3. Static size variable areas
   (GLOBALS and OWNS)

4. Stacks

Programs are 'pure' - they do not unintentionally modify themselves. Programs and constant areas can be placed in contiguous read only memory. Static variable storage and stack areas must be placed in read/write memory.

4.2 REGISTERS

The sixteen registers are divided into three main classes:

1. Reserved registers:

   These registers are declared in the module head. Their scope is the entire module (they may be accessed from within any routine). They are never saved.

2. BLISS-10 run-time registers:

   After the reserved registers have been allocated, the lowest three remaining addresses are assigned as the run-time registers unless specified by XREG= in the module head. In particular, if there are no reserved registers, 0, 2, and 3 are assigned as the S, F, and V registers respectively. The names SREG, FREG, and VREG are available at the outermost blocks of the module and, as in the case of reserved registers, these names are accessible from within any routine.

   Note that only SREG can be defined as register 0; a warning message is otherwise generated and the declaration is ignored.

3. Temporary registers:

All the remaining registers fall into this class and are divided into two subclasses:

a. savable:

These registers are used for declared registers, control registers in incr-decr loops, and when necessary for computing temporary values. Any of these registers used in the body of a function or routine is saved in the prolog and restored in the epilog. Of course if F is not a global routine and F is within the scope of register R, then R is not preserved. The user must declare the size of this block of registers in the module head. (DREGS =). These registers are allocated from the highest addresses.

b. non-savable:

These are the registers used for calculating intermediate results. They are saved at the call site of a function or routine only if they contain a needed result and are never saved in the prolog or epilog.

Comments:

1. If one wishes to load a collection of BLISS-10 modules together, they must request precisely the same reserved registers and request the same number of savable temporaries.

2. The two classes of temporary registers are managed quite differently in that the savable registers obey a stack discipline (to minimize saving and restoring) and the non-savable are used in round-robin fashion (to lengthen the life of intermediate results). The present version of the compiler requires a minimum of 4 non-savable registers--i.e., the maximum value of DREGS = 9 - # of reserved regs. In general the compiler can produce better code if DREGS is kept to the minimum value which the lexical scope of declared registers and/or incr-decr loops allow.

4.3 THE STACK AND FUNCTIONS

The first 17 locations of each stack are reserved for state information (registers plus program counter) for a process when it is inactive. The configuration above these 17 state words depends upon the depth of nesting of function calls, but each such nested call involves a similar (not identical) use of the stack. Figure 4-1 illustrates a typical stack configuration after several nested functional calls. At a time when one of these functions is executing

1. The S-register points to the highest assigned cell in the stack; the S-register is used to control the allocation of the stack area.

2. The F-register points to the 'local base of stack'. Below(1) the F-register are the parameters to the function and the return address. The stack cell actually pointed to by the F-register contains the previous value of the F-register at the time at which the current function was entered.

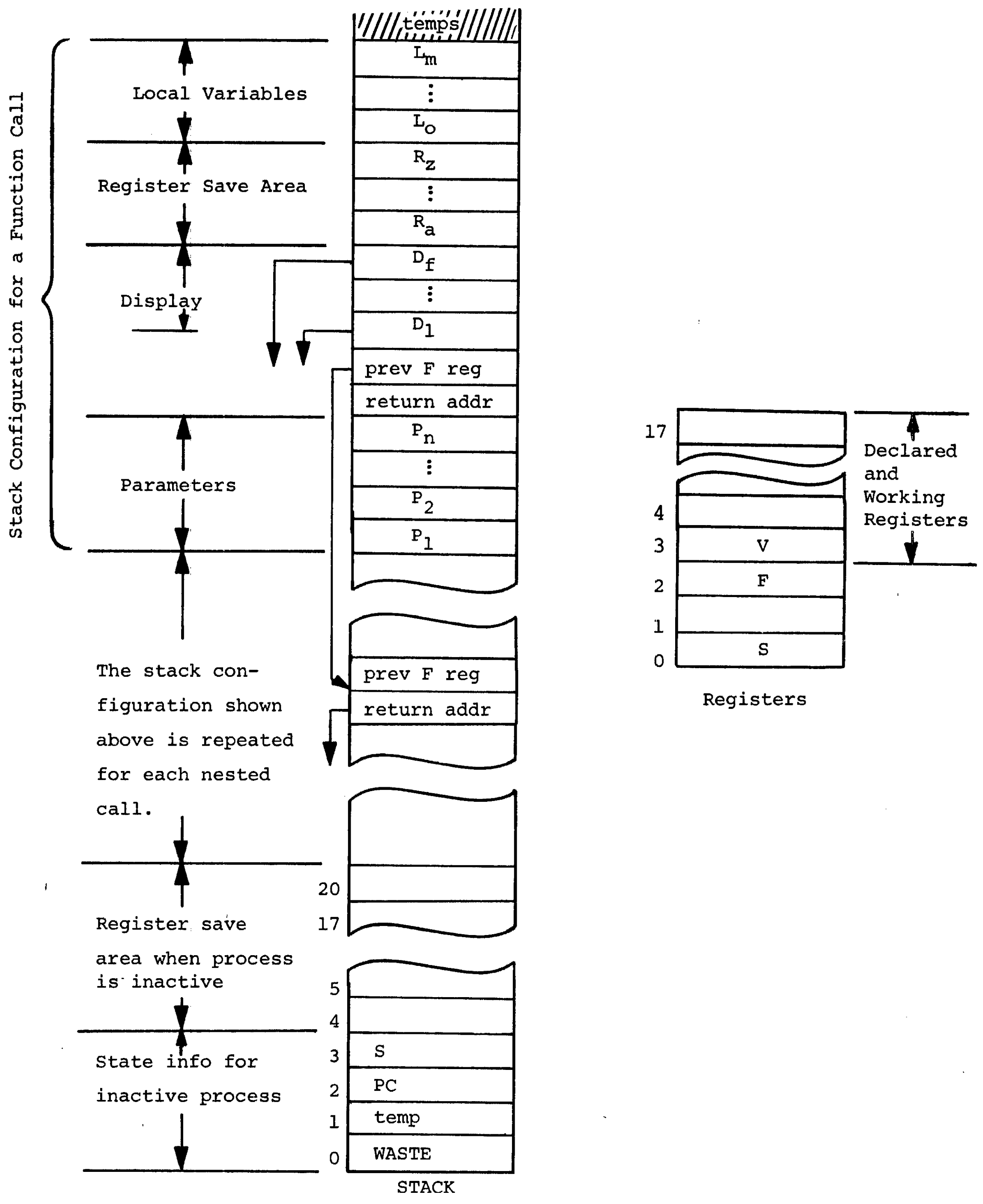
3. The calling sequence which is used to enter a function (or routine) is

```
    PUSH     S,p1          ;   push 1st parameter onto the
                               stack

    PUSH     S,p2          ;   push 2nd parameter onto the
                               stack

     ...                  ...

    PUSH     S,pn          ;   push nth parameter onto the
                               stack

    PUSHJ    S,FCN         ;   jump to the called function

    SUB      S,[n,,n]      ;   delete the parameters
```

4. Above the F-register are stored the "displays", D1...Df. One display is used for each lexically embracing FUNCTION declaration. The value of each display is the F-register value for the most recent recursive entry for that lexically embracing function. The displays are needed and used to access variables global to the current function but local to embracing function. Such access is prohibited in routines, and consequently no displays are saved on a routine entry.

----

(1)'below' in the sense of decreasing address values.



Stack Configuration for a Function Call
Local Variables
Register Save Area
Display
Parameters
The stack configuration shown above is repeated for each nested call.
Register save area when process is inactive
State info for inactive process
temps
L_m
L_o
R_z
R_a
D_f
D_1
prev F reg
return addr
P_n
P_2
P_1
prev F reg
return addr
20
17
5
4
3
2
1
0
S
PC
temp
WASTE
STACK
17
4
3
2
1
0
V
F
S
Registers
Declared and Working Registers

Figure 4-1
Typical Stack Configuration

5. Above the displays are saved any savable registers which are destroyed by the execution of the function body. These registers are restored before the function exits.

6. Any local variables in the function are stored on top of the saved registers. Space is acquired/deleted for locals on block entry/exit by simply adding/subtracting a constant to the S-register. Some of these locals are automatically generated by the compiler, for example: An excessive number of declared registers, or the evaluation of an extremely complex expression exhausting the available registers, may force the area above the locals to be used for storing partial results of an expression evaluation.

7. The V-register is used to return the value of the function or routine.

Figure 4-2 illustrates the code generated surrounding the body of a function. The code surrounding a routine body is identical, with the exception that the displays are never saved. In this illustration the S, F, and V registers are shown occupying physical registers 0,2 & 3. In practice other registers may be chosen if these registers are reserved in the module head.

Function Prolog and Epilog

```
FCN: PUSH   S,F        ;    save old F-register
     PUSH   S,1(F)     ;    copy display zero
     ...               ...
     PUSH   S,f(F)     ;    copy display f
     HRRZ   F,S        ;    set up new F
     SUBI   F,f        ;    subtract no. displays
     PUSH   S,F        ;    new display created
     PUSH   S,Ra       ;    save register
     ...               ...                                Not
     PUSH   S,Rz       ;    save register                 Generated
                                                        --For
         BODY OF FUNCTION OR ROUTINE                      Routines


     POP    S,Rz       ;    restore register
     ...               ...
     POP    S,Ra       ;    restore register
     SUB    S,[(f+1),,(f+1)] ; eliminate displays
     POP    S,F
     POPJ   S,
```

Figure 4-2
Function Prolog and Epilog

Block Entry and Exit

```
BENTER:   MOVEM   R1,1+1(F)           ; save    in-use    working
                                        registers
          ...                         ...
          MOVEM   Rj,1+j(F)           ; save    in-use    working
                                        registers
          ADD     S ,[n,,n]           ; INCR   S-register   by   no.
                                        locals in blk

BEXIT:    SUB     S,[(n+j),,(n+j)]    ; DECR   S-register   by   no.
                                        locals in blk
                                      ; (note:  inuse reg's left  in
                                        stack,
                                      ; re-loaded only when used)
```

Figure 4-3
BLOCK ENTRY and EXIT

4.4 ACCESS TO VARIABLES

This section briefly indicates the mechanism by which generated code accesses various types of variables (formals, owns and globals, locals, etc.) the exact addressing scheme used by the compiler in any particular case is highly dependent upon the context; however, the following material should aid in understanding the overall strategy.

1. OWN and GLOBAL variables are accessed directly.

2. Formal parameters of the current routine are accessed negatively with respect to the F-register. If the current routine has n formals, then the ith one is addressed by

   (-n + i - 2)(F)

3. Local variables of the current routine are accessed positively with respect to the F-register. To access the ith local cell, one uses

   (i + d + r + 1)(F)

   where d is the number of displays saved and r is the number of registers saved on function entry.

4. Formal parameters and local variables which are not declared in the currently executing function are accessed through the display. The appropriate display is copied into one of the working registers then accessed by indexing through that register in a manner similar to that shown in (2) or (3) above.

The first four characters of the name introduced in the module head is used to name various regions in the produced code. These names are declared global and therefore are available in DDT. If 'XXXX' are the first four characters of the module name, then

XXXX.C	is the location of the start of the constant pointer area which contains relocatable constants generated by the compiler.
XXXX.0	is the location of the first code word (not necessarily starting address) of this module.
XXXX.F	is the location of the first instruction in the main body of the module, i.e., it is the starting address of the module.
XXXX.L	is the location of the "literal" area which contains compile-time, unrelocated constants generated by the compiler.
XXXX.O	is the location of the "own" area in which is stored all variables declared 'own' in the module.
XXXX.G	is the location of the "global" area in which is stored all variables declared "global" in the module.

XXXX.. is the module name (recognized by DDT).

XXXX.P is the first location of the "plit" area.

Note that with (normal) two-segment conventions, XXXX.C, XXXX.0, XXXX.F, XXXX.L, XXXX.., and XXXX.P will be high-segment addresses while XXXX.O and XXXX.G will be low-segment addresses.

4.5 MAIN PROGRAM CODE

4.5.1 CCL Entry Linkage

If the "CCL" switch is declared in the module head, the following two instructions are generated as the first two executable instructions:

```
TDZA  $V,$V
MOVEI $V,1
```

Linkage via a RUN UUO with an offset of 1 causes entry to the second instruction of the program (i.e., the MOVEI) whereas normal entry (via a RUN, R, START command) is to the first instruction. The first evaluated expression in the program must be of the form "X ← .VREG" [assuming the user has not declared the name VREG, and it retains the meaning of the value register], which stores the value set by one of the two instructions. A "0" (false) indicates normal entry while a "1" (true) indicates a CCL-type entry.

4.5.2 Stack Initialization

If a STACK declaration occurred in the module head, the following code is generated to initialize the F, S, and B registers. This code follows the CCL entry code, if any (see 4.5.1).

```
HRRZI  $F, stack-address
MOVEM  $F, .BREG
HRLI   $S, <[stack-length]-[coroutine-prefix-length]>
HRRZI  $F, [coroutine-prefix-length]($F)
HRR    $S, $F
```

4.5.3 Program Termination

All modules terminate with the instruction

```
CALLI  0,#12
```

which is the EXIT UUO for the DECsystem-10 monitors. This is the standard terminating execution of user programs.

4.6 TIMER CODE

If there is a TIMER declaration in the module head, and it has been activated by the /T switch in the command string or the TIMING switch in the module head, the routine calls below are generated. The entry routine-call precedes the standard prolog code (see figure 4-2), and the exit routine-call precedes the standard epilog code.

```
entry call:     TIMER-routine (routine-descriptor<0,0>)

exit call:      TIMER-routine
                ((-1↑18) or routine-descriptor<0,0>)
```

The actual code generated is:

```
     entry call:     HRRZI    R, routine-descriptor
                     PUSH     $S, R
                     PUSHJ    $S, timer-routine
                     SUB      $S, [000001,,000001]

     exit call:      PUSH     $S, $V
                     HRROI    R, routine-descriptor
                     PUSH     $S, R
                     PUSHJ    $S, [000001,,000001]
                     POP      $S, $V
```

Note that the value of the V-register is preserved across the call. Hence any value returned by the timer-routine is lost.

As of this writing, no routine-descriptor is produced, and the address of the routine-descriptor is always 0.

If the module contains a STACK declaration, then additional code is generated in the main body of the program. An entry call is generated following stack initialization, and an exit call is generated immediately preceding the CALLI which terminates the program. If the CCL switch occurs in the module head, additional code is generated to save the V-register across the entry call.

CHAPTER 5
BLISS-10 EXAMPLES

This section contains a set of examples which illustrate the use of BLISS-10. Each example is intended to be fairly complete and self contained and to illustrate one or more features of the language.

The authors would like to invite others to contribute further examples for inclusion in this section. New examples will be included if they clearly illustrate features and/or uses of the language which are not already adequately illustrated.

Example 1: A TT-Call I/O Package

The following set of declarations defines a set of Teletype input/output routines using the DECsystem-10 monitor TT-call mechanism. The set of functions is not complete, but adequate to illustrate the approach.

The declarations below provide the following functions:

INC	Input one character - wait for EOL before returning.
OUTC	Output one character.
OUTSA	Output ASCIZ-type string beginning at specified address.
OUTS	Output ASCIZ-type string specified as the parameter.
OUTM	Output multiple copies of a specified character.
CR	Output carriage return.
LF	Output line feed.
NULL	Output null character.
CRLF	Output carriage return and line-feed followed by 2 nulls.
TAB	Output tab.
OUTN	Output number in specified base and minimum number of digits.
OUTD	Output decimal number with at least one digit.
OUTO	Output octal number with at least one digit.
OUTDR	Output decimal number with at least specified number of digits.

OUTOR   Same as OUTDR except octal.

```
MODULE TTIO(STACK)=BEGIN

MACHOP TTCALL=#51;

MACRO   INC= (REGISTER Q; TTCALL(4,Q); .Q)$,
        OUTC(Z)= (REGISTER Q; Q←(Z); TTCALL(1,Q))$,
        OUTSA(Z)=TTCALL(3,Z)$,
        OUTS(Z)= OUTSA(PLIT ASCIZ Z)$,
        OUTM(C,N)= DECR I FROM (N)-1 TO 0 DO OUTC(C)$,
        CR= OUTC(#15)$, LF= OUTC(#12)$, NULL= OUTC(0)$,
        CRLF= OUTS('?M?J?0?0')$,
        TAB= OUTC(#11)$;


ROUTINE OUTN(NUM,BASE,REQD)=
  BEGIN OWN N,B,RD,T;
    ROUTINE XN=
      BEGIN LOCAL R;
        IF .N EQL 0 THEN RETURN OUTM("0",.RD-.T);
        R←.N MOD .B; N←.N/.B; T←.T+1; XN();
        OUTC(.R+"0")
      END;

    IF .NUM LSS 0 THEN OUTC("-");
    B←.BASE; RD←.REQD; T←0; N←ABS(.NUM); XN()
  END;


MACRO   OUTD(Z)= OUTN(Z,10,1)$,
        OUTO(Z)= OUTN(Z,8,1)$,
        OUTDR(Z,N)= OUTN(Z,10,N)$,
        OUTOR(Z,N)= OUTN(Z,8,N)$;
```

```
!       THE PROGRAM BELOW PRINTS A TABLE OF INTEGERS, THEIR
!       SQUARES, AND THEIR CUBES:

    OWN N,C;

    CRLF; OUTS('INPUT AN INTEGER PLEASE ...');
    N←0; WHILE (C←INC) GTR "0" AND .C LSS "9" DO
          N=.N*10+(.C-"0");

    CRLF; OUTS('A TABLE OF THE SQUARES AND CUBES OF 1-');
            OUTD(.N);

    CRLF; INCR I FROM 1 TO 3 DO (TAB; OUTS(' X↑');
            OUTD(.I));

    CRLF; INCR I FROM 1 TO 3 DO (TAB; OUTM("-",5));

    INCR I FROM 1 TO .N DO
      BEGIN OWN X;
        X←.I; CRLF;
        DECR J FROM 2 TO 0 DO (TAB; OUTD(.X); X←.X*.I)
      END;

END ELUDOM
```

Although the example is quite simple, there are several things about it which should be noted:

1. The use of a MACHOP declaration and embedded assembly code.

2. The use of macros to add a level of "syntactic sugar" and general cleanliness to the code.

3. The use of the escape character "?" in the CRLF macro to obtain control characters (e.g., carriage-return) in strings.

4. Parenthesization of macro parameters, as in OUTM, to insure proper hierarchy relations in the expansion.

5. The use of "DECR-TO-ZERO" in OUTM because it produces better code than "INCR-TO-EXPRESSION".

6. The use of own variables and the parameterless procedure XN in OUTN in order to avoid passing redundant parameters through the recursive levels of XN.

7. The fact that the local variable "R" is local to each recursive level of XN and hence its value is preserved at each level.

Example 2: Queue Management Model

This module contains routines to insert and delete items on doubly-linked queues. In addition it contains space management routines implementing the "Buddy System" (cf: Knuth: Vol. 1).

Buddy System

This is not intended to be a detailed description of the buddy system model of space management. We will simply give a brief description of this implementation of the scheme. The vector of allocatable space is called MEM. Space is allocated and deallocated from MEM by the routines GET and RELEASE, respectively. The basic unit of allocatable space is an item. Items are of size 2**ITEMSIZE where 0 < ITEMSIZE <= LOG2MEMSIZE. The first two words of an item are formatted:

ITEMSIZE	RLINK
<NOT-USED>	LLINK

Available items of size N are elements of a doubly linked list whose header is the two word cell SPACE[N]. The routines LINK and DELINK are called to enter and remove items from lists. The routine COLLAPSE is used to compact two adjacent available items of size 2**N into an item of size 2**(N+1). The COLLAPSE routine iterates this process until no more compaction can take place.

Queue Model

In this model a queue is defined to be a doubly-linked list suspended m a header whose first three words are formatted as follows:

0 17|18 36

HEADER SIZE	RLINK
(not used)	LLINK
REMOVE	ENTER

The fields REMOVE and ENTER contain the addresses of the routines to be invoked when removing and entering items on the queue. To enter item X on queue Q, one simply makes the call ENQ(X,Q). ENQ then invokes the enter routine in Q's header which returns the address of the item in Q after which X is to be inserted. In a similar manner one removes the "next" item from queue Q by the call DEQ(Q). DEQ then invokes the remove routine in Q's header to return the address of the "next" item. The advantage of this scheme is that the queueing discipline is queue specific, and the same primitives (ENQ and DEQ) may be used independent of the discipline used for that queue. Examples of the enter and remove routines for LIFO, FIFO, and PRIORITY type queues appear at the end of this example module.

```
MODULE QMS(STACK)=

! BUDDY SYSTEM
!---------------
```

```
BEGIN

BIND MEMSIZE=1↑12;

GLOBAL VECTOR MEM[MEMSIZE];
BIND LOGZMEMSIZE=35-FIRST-ONE(MEMSIZE);
STRUCTURE ITEM[I,J,P,S]=
        CASE .I OF
          SET
            (.ITEM)<.P,.S>;
            (@.ITEM+.J)<.P,.S>;
            (@@.TIEM+.J)<.P,.S>;
            (@(@.ITEM+1)+.J)<.P,.S>
          TES;

STRUCTURE VECTOR2[I]=
        [2*I](.VECTOR2+2*.I)<0,36>;

MACRO   BASE=0,0,0,18$,
        RLINK=1,0,0,18$,
        LLINK=1,1,0,18$,
        ITEMSIZE=1,0,18,18$,
        NXTRLINK=2,0,0,18$,
        NXTLLINK=2,1,0,18$,
        PRVRLINK=3,0,0,18$,
        PRVLLINK=3,1,0,18$,

GLOBAL VECTOR2 SPACE[LOG2MEMSIZE+1];

BIND VECTOR SIZE =
        PLIT(1↑0,1↑1,1↑2,1↑3,1↑4,1↑5,1↑6,1↑7,1↑8,1↑9,1↑10,
             1↑11,1↑12);

MACRO   PARTNER(B1,B2,S)= ((((B1)-MEM<0,0>)
        XOR ((B2)-MEM<0,0>))
                              EQL .SIZE[S])$,
        REPEAT= WHILE 1 DO$,
        BASEADDR(B,S)= MEM[((B)-MEM<0,0>)
        AND NOT .SIZE[S]]<0,0>$,
        ERRMSG(S)= ERROR(PLIT ASCIZ S)$;



! SPACE-MANAGEMENT-ROUTINES
!-------------------------------

FORWARD EMPTY,ERROR,LINK,DELINK,COLLAPSE;

GLOBAL ROUTINE GET(N)=

  !RETURNS THE ADDRESS OF AN ITEM OF SIZE 2**N

  BEGIN REGISTER ITEM R;
    IF .N LEQ 0 OR .N GTR LOG2MEMSIZE
      THEN ERRMSG('INVALID SPACE REQ');
    IF NOT EMPTY(SPACE[.N<0,0>)
      THEN R[BASE]←DELINK(.SPACE[.N])
      ELSE
```

```
        BEGIN
          R[BASE]←GET(.N+1);
          COLLAPSE(.R[BASE]+.SIZE[.N],.N)
        END;
    R[ITEMSIZE]←.N;
    .R[BASE]
  END;

ROUTINE COLLAPSE(A,N)=

  !CALLED BY RELEASE AND GET TO ATTEMPT TO COMPACTIFY SPACE
  !IF ADJACENT ITEMS ARE FREE

  BEGIN MAP ITEM A; REGISTER ITEM L; LABEL CYCLE;
    REPEAT
      CYCLE: BEGIN
        L[BASE]←SPACE[.N]<0,0>;
        WHILE .L[RLINK] NEQ SPACE[.N]<0,0> DO
          IF PARTNER(.L[RLINK],.A[BASE],.N)
            THEN
              BEGIN
                A[BASE]←BASEADDR(DELINK(.L[RLINK]),.N);
                N←.N+1;
                LEAVE CYCLE
              END
            ELSE L[BASE]←.L[RLINK];
        RETURN (A[ITEMSIZE]←.N; LINK(.A[BASE].,L[BASE]))
      END;
  END;

GLOBAL ROUTINE RELEASE(A)=

  !CALLED TO RELEASE ITEM A

  BEGIN
    MAP ITEM A;
    COLLAPSE(.A[BASE].,A[ITEMSIZE])
  END;


! SIMPLE-LIST-ROUTINES
!-----------------------

ROUTINE DELINK(A)=

  !REMOVES ITEM A FROM THE LIST TO WHICH IT IS APPENDED

  BEGIN MAP ITEM A;
    A[PRVRLINK]←.A[RLINK];   A[NXTLLINK]←.A[LLINK];
    A[RLINK]←A[LLINK]←.A[BASE]
  END;

ROUTINE LINK( I,TOO)=

  ! INSERTS ITEM A INTO A LIST IMMEDIATELY AFTER THE ITEM
  ! TOO
```

```
    BEGIN
      MAP ITEM A:TOO;
      A[LLINK]←.TOO[BASE];  A[RLINK]←.TOO[RLINK];
      TOO[NXTLLINK]←TOO[RLINK]←.A[BASE]
    END;

  ROUTINE RELINK(A,TOO)=

    ! REMOVES ITEM FROM ITS PRESENT LIST AND INSERTS IT AFTER
    ! TOO

    LINK(DELINK(.A),.TOO);


  ROUTINE EMPTY(L)=

    !PREDICATE INDICATING EMPTY LIST

    BEGIN MAP ITEM L;
      .L[BASE] EQL .L[RLINK]
    END;
```

```
! QUEUE-HANDLING-ROUTINES
!--------------------------

MACRO   QHDR-ITEMS;

MACRO   ENTER=1,2,0,18$,
        REMOVE=1,2,18,18$;


GLOBAL ROUTINE ENQ(A,Q)=

  ! ENTERS ITEM A ON QUEUE Q ACCORDING TO THE INSERTION
  ! DISCIPLINE EVOKED BY Q'S ENTER ROUTINE

  BEGIN
    MAP QHDR Q;
    RELINK(.A,(.Q[ENTER])(.Q[BASE],.A))
  END;


GLOBAL ROUTINE DEQ(Q)=

! REMOVES AN ITEM FROM QUEUE Q ACCORDING TO THE REMOVAL
  ! DISCIPLINE EVOKED BY Q'S REMOVE ROUTINE

  BEGIN
    MAP QHDR Q;
    DELINK((.Q[REMOVE])(.Q[BASE]))
  END;


! MISC SERVICE ROUTINES
!-----------------------

ROUTINE ERROR(A)=
  BEGIN MACHOP TTCALL=#051;
    TTCALL(3,.A)
  END;


ROUTINE INITIALIZE=

  !INITIALIZES THE SPACE MANAGEMENT DATA

  BEGIN REGISTER ITEM X;
    X[BASE]←MEM<0,0>;
    X[RLINK]←X[LLINK]←SPACE[LOG2MEMSIZE]<0,0>;
    X[ITEMSIZE]←LOG2MEMSIZE;
    DECR I FROM LOG2MEMSIZE-1 TO 0 DO
      SPACE[.I]←(SPACE[.I]+1)<0,36>←SPACE[.I]<0,0>;
    SPACE[LOG2MEMSIZE]←(SPACE[LOG2MEMSIZE]+1)<0,36>←MEM<0,0>
  END;


! EXAMPLES OF VARIOUS QUEUE MODELS
!-----------------------------------
```

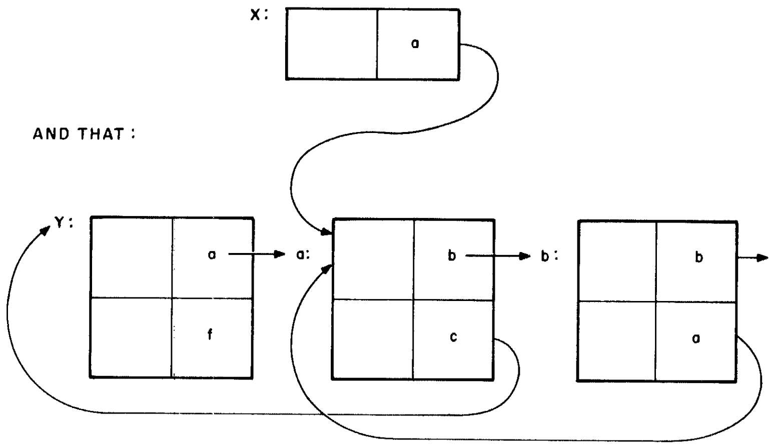
```
! LIFO QUEUE
!-------------

ROUTINE LIFOREMOVE(Q)=
  BEGIN
    MAP QHDR Q;
    IF EMPTY(.Q[BASE]) THEN
      ERRMSG('INVALID DEQ REQUEST');
    .Q[RLINK]
  END;

ROUTINE LIFOENTER(Q,A)=
  BEGIN
    MAP QHDR Q;
    .Q[BASE]
  END;


! FIFO QUEUE
!-------------


ROUTINE FIFOREMOVE(Q)=
  BEGIN
    MAP QHDR Q;
    IF EMPTY(.Q[BASE]) THEN
      ERRMSG('INVALID DEQ REQUEST');
    .Q[RLINK]
  END;


ROUTINE FIFOENTER(Q,A)=
  BEGIN
    MAP QHDR Q;
    .Q[LLINK]
  END;


! PRIORITY QUEUE
!-----------------


MACRO   PRIOIRITY=1,1,18,18$;

ROUTINE PRIREMOVE(Q)=
  BEGIN
    MAP QHDR Q;
    IF EMPTY(.Q[BASE]) THEN
      ERRMSG('INVALID DEQ REQUEST');
    .Q[RLINK]
  END;


ROUTINE PRIENTER(Q,A)=
  BEGIN
    MAP QHDR A; MAP ITEM A; REGISTER ITEM L;
    IF EMPTY(.Q[BASE]) THEN RETURN .Q[BASE];
    L[BASE]←.Q[LLINK];
    UNTIL .L[PRIORITY] GEQ .A[PRIORITY] DO
      L[BASE]←.L[LLINK];
    .L[BASE]

  END ELUDOM
```

Comments on the Use of BLISS in the Implementation

1. The structure ITEM is particularly interesting and perhaps at first a bit obscure.

   To illustrate, consider a variable X structured by item:

   Assuming that the right half of X contains a:



   Then:

```
        .X[BASE]   == a        .X[NXTRLINK]  = d
        .X[RLINK]  == b        .X[NXTLLINK]  = a
        .X[LLINK]  == c        .X[PRVRLINK]  = a
                               .X[PRVLLINK]  = f
```

   The structure ITEM uses the "constant case" expression to distinguish between the pointer, the pointee, and the pointee's predecessor and successor.

2. The structure VECTOR2 has a size expression [2*I] which is used in the allocating declaration:

```
        GLOBAL VECTOR2 SPACE[LOG2MEMSIZE+1];
```

3. Since the addresses of the 'remove' and 'enter' routines are stored in the queue header, the expression

```
        (.Q[REMOVE])(.Q[BASE])
```

is a call of the routine whose address is .Q[REMOVE] and passes it to the base address of the queue or its parameter.

4. The macro 'REPEAT = WHILE 1 DO' defines an infinite loop - its only exit is defined by the RETURN expression in its body.

5. Notice the 'BIND VECTOR SIZE = PLIT(1↑0,1↑1,1↑2,...' in the space allocator. The value of SIZE is a pointer to this sequence of values, and in particular the value of '.SIZE[.N]' is 2↑N.

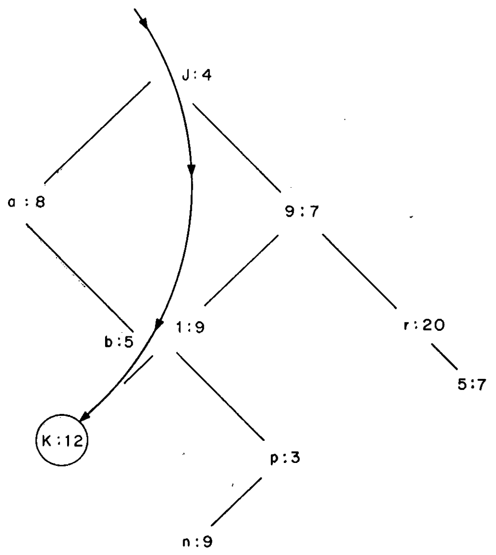
Example 3: Discrimination Net

A discrimination net is a mechanism used to associate "information" with "names". The net is actually a tree, each node of which consists of a name and the information associated with that name, as well as a set of pointers to other nodes. To look up a name in the net we start at the root node and see if the name in the node matches our target name. If it does, we return the associated information.

Otherwise, we use a "discrimination function" which determines which subnode to examine next (usually as a function of the target name and the name of the current node). If there is no corresponding subnode, a new node must be created.

For example, a binary net (two subnodes/node) with a discrimination function which chooses the left branch if the target name is alphabetically smaller than the name in the node, is illustrated below:

Name:	j,	9,	1,	a,	b,	r,	p,	n,	s,	k
Inf:	4,	7,	9,	8,	5,	20,	3,	9,	7,	12



In the implementation which follows, there are three globally defined routines:

1. DSCINIT (String address) -- returns a pointer to the information field of the node associated with the string. This must be called first to initialize the net. (The information field will be zeroed when the node is new.)

2. DSCLKP (String address) -- the "lookup" routine. Value returned as above.

3. DSCPNAME (Information field address) -- returns a pointer to the print name associated with the particular information field.

The implementation is designed to allow the user to create a module somewhat "tailored" to his needs. The module is created by passing:

1. the estimated number of entries to be inserted into the table;

2. the average number of words each name will occupy;

3. the number of words in the "information field";

4. the number of subnodes of each node (e.g., binary example above, 2);

5. a string which executes an error routine

in that order, to a macro "DSCRIMINET". Two macros must be defined previous to the DSCRIMINET expansion:

1. DSCIMINATE (Target string address, current node string address) must have a value of -1 if the strings match. Otherwise, its value must be between 0 and 1 less than the number of subnodes.

2. DSCCOPY (To address, From address) copies the string from the "from address" to the "to address", returning the number of words occupied by the copy.

```
MODULE NET(STACK=GLOBAL(STABK,#400))=
BEGIN
MACRO
  DSCRIMINET(MAXNUMENT,AVNAMESIZE,INFSIZE,NOSUBNODES,ERROR)=

    BEGIN
      %N.B.: ALL VECTOR ACCESSES INDIRECT THROUGH BASE%
      STRUCTURE VECTOR[I]=(.VECTOR+.I)<0,36>;

      % NET SPACE ALLOCATION, STRUCTURE DEFINITION AND
        INITIALIZATION DEFINITIONS %
      BIND TABLELEN=MAXNUMENT*((NOSUBNODES+1)/2+INFSIZE
           +AVNAMESIZE);
      OWN BASENODE[TABLELEN];
      BIND MAXADD=BASENODE+TABLELEN;

      BIND SUBNODE=0, INF=1, PNAME=2,
           INFOFFSET=(NOSUBNODES+1)/2,
           PNAMEOFFSET=INFOFFSET+INFSIZE;

      STRUCTURE NODE[SUBFIELD,INDEX]=CASE .SUBFIELD OF
        SET .NODE[.INDEX↑(-1)]<IF .INDEX THEN 18,18>;
            .NODE[INFOFFSET];
            .NODE[PNAMEOFFSET] TES;

      GLOBAL ROUTINE DSCPNAME(INFPOS)=
        (.INFPOS+INFSIZE)<0,36>;

      OWN NODE NEXTCELL;

      ROUTINE INITNODE(CELL,STRING)=
        BEGIN
          DECR I FROM PNAMEOFFSET-1 TO 0 DO CELL[.I]←0;
          IF MAXADD LEQ (NEXTCELL=.NEXTCELL+PNAMEOFFSET+
            (MAP NODE CELL; DSCCOPY(CELL[PNAME],.STRING)))

            THEN ERROR ELSE .CELL
        END;

      GLOBAL ROUTINE DSCINIT(STRING)=
        BEGIN
          LOCAL NODE RETVAL;
          NEXTCELL=BASENODE;
          RETVAL=INITNODE(BASENODE,.STRING);
          RETVAL[INF]
        END;

      ROUTINE NEWCELL(STRING)=INITNODE(.NEXTCELL,.STRING);

      % THE LOOKUP ROUTINE ITSELF %
      GLOBAL ROUTINE DSCLKP(STRING)=
        BEGIN
          LOCAL DISCIND, NODE CURRENT:NEXT;
          NEXT←BASENODE;

          DO
            BEGIN
              CURRENT←.NEXT;
              IF (DISCIND←DSCIMINATE(.STRING,
                  CURRENT[PNAME])) LSS 0
                THEN RETURN CURRENT[INF];
              NEXT←.CURRENT[SUBNODE,.DISCIND]
            END

            UNTIL .NEXT EQL 0;

          NEXT←CURRENT[SUBNODE,.DISCIND]←NEWCELL(.STRING);
          NEXT[INF]
        END;
    END;$;


  ROUTINE DESCIMINATE(L,R)=
    BEGIN
      STRUCTURE VECTOR[I]=(@.VECTOR+.I)<0,36>; LABEL LOOP;
      LOOP: INCR I FROM 0
        DO BEGIN
             BIND LEFT=.L[.I], RIGHT=.R[.I];
             IF LEFT NEQ RIGHT THEN
                LEAVE LOOP WITH (LEFT LSS RIGHT);
             IF (LEFT AND #376) EQL 0 THEN
                LEAVE LOOP WITH  -1
           END
    END;

  ROUTINE DSCCOPY(INTO,FRO)=
    BEGIN
      STRUCTURE VECTOR[I]=(@.VECTOR+.I)<0,36>; LABEL LOOP;
      LOOP: INCR I FROM 0 DO
        IF ((INTO[.I]=.FRO[.I]) AND #376) EQL 0
          THEN LEAVE LOOP WITH .I+1
    END;

  EXTERNAL ERROR;
  DSCRIMINET(500,3,1,2,ERROR(PLIT 'LOOKUP TABLE OVERFLOW'))

BEGIN
  BIND NAMES=PLIT(
    PLIT ASCIZ 'FIRSTNAME',
    PLIT ASCIZ 'SECOND',
    PLIT ASCIZ 'SS',
    PLIT ASCIZ 'A LONGISH NAME',
    PLIT ASCIZ 'L',
    PLIT ASCIZ '77788()34');

  EXTERNAL DSCLKP, DSCINIT;
  DSCINIT(PLIT 'ZEROTH NAME')=-3;
  INCR I FROM 0 TO .NAMES[-1]-1 DO DSCLKP(.NAMES[.I])=.I;
  INCR I FROM 0 TO .NAMES[-1]-1 BY 2 DO DSKLKP(.NAMES[.I])=.I+1↑35;
END;
END ELUDOM;;
```

Notes on the Implementation

The BLISS module above implements the example described at the beginning of this section. The test program portion of the module simply initializes the table, inserts the six strings in the plit into the table (associating as information, the index in the plit), and runs through the evenly indexed items in the plit, turning on the sign bit in the information word.

Of interest:

1. The vector structure (which defaults as the structure for all unmapped variables and expressions) is redefined "indirectly"; this is fairly dangerous in any program, and represents an after-the-fact programming decision.

2. The physical structure of the table is kept independent of the logical structure as used by the lookup routine; no reference is made from the lookup routine to the structure other than through the structured nodes.

3. The binds, structures, own declarations and even the initialization function - requiring knowledge of the physical structure - are kept grouped and separate. Note, for example, that INITNODE uses both a vector mapping on contiguous fields of CELL and NODE structure.

4. The physical structure of the tree is kept isolated from the user of the routines to the extent that only knowledge that the mechanism is associative is of importance -- the particular lookup algorithm and storage management are independent of the functional use of the module.

5. BLISS-10 programming "tricks":

   a. Use of the constant case expression for subfields of structures (NODE in this case);

   b. Default use of 0 for the omitted else in the structure case defining the SUBNODE field;

   c. CELL remapped in the INITNODE routine to take advantage of knowledge of the physical layout of the NODE's storage.

   d. "Dynamic" binds of LEFT and RIGHT inside the loop in the test version discrimination function;

   e. The bind to a plit (of NAMES) in the test portion, to prevent duplicate storage allocation for the twice-used plit;

   f. Stores into routine cells in the test program loops;

   g. Use of the plit length word preceding the plit (NAMES[-1]).

Example 4: Simple Monitor I/O

These routines supply low-level support for programs which run in the user mode under the DECsystem-10 timesharing monitor. Although not all I/O facilities are shown in this example, the same methods may be used to provide any I/O facility in BLISS-10.

These routines compile to about 75 words of code, and form about the smallest set of routines required to read and write files. The restriction they impose is that there may be only one input file and one output file, and the channels associated with these files must be specified at compile time.

```
MODULE IO(ENTRIES=(LOOKUP,ENTER,OPENIN,OPENOUT,CLOSEIN,
                CLOSEOUT,PURGEOUT,OUTMSG,READ,WRITE))=
BEGIN
%%
%

             THIS MODULE PROVIDES SOME I/O EXAMPLES USING
        BLISS-10.  FOR THIS SET OF EXAMPLES, WE WILL ASSUME A
        STATIC CHANNEL ASSIGNMENT.  ALL INPUT WILL BE ON
        CHANNEL "INCH", AND ALL OUTPUT ON CHANNEL "OUTCH".
        THE THREE-WORD BUFFER HEADER AREAS WILL BE "IBUFH"
        AND "OBUFH" AND ARE DECLARED GLOBAL IN SOME OTHER
        MODULE.
%
%%

BIND
        INCH=1,
        OUTCH=2,

%%
%
        HERE ARE SOME USEFUL MACHINE OPERATIONS
%
%%

MACHOP
        CALLI=#047,
        OPEN=#050,
        TTCALL=#051,
        IN=#056,
        OUT=#057,
        GETSTS=#062,
        STATZ=#063,
        CLOSEUUO=#070,
        RELEAS=#071,
        LOOKUPUUO=#076,
        ENTERUUO=#077,
        XCT=#256;

%%
%
```

```
        THE FOLLOWING MACRO RETURNS "TRUE" IF THE
        EXPRESSION GIVEN AS ITS PARAMETER SKIPS, "FALSE"
        IF NOT
%
%%

MACRO SKIP(OP)=IFSKIP (OP) THEN 1 ELSE 0$;

                        %%
%
        HERE ARE SOME USEFUL MACROS
%
%%

MACRO
        RESET=CALLI(0)$,
        COUNT(BUFH)=BUFH[2]$,
        PTR(BUFH)=BUFH[1]$;

%%
%
             IBUFH AND OBUFH MUST BE DECLARED GLOBAL IN
        ANOTHER MODULE
%
%%

EXTERNAL
        IBUFH[3],
        OBUFH[3];

%%
%

             "OPENIN" AND "OPENOUT" TAKE THREE PARAMETERS:
        THE DEVICE STATUS (INCLUDING DATA MODE), THE LOGICAL
        DEVICE NAME, AND THE BUFFER POINTERS, PRECISELY AS
        SPECIFIED FOR THE OPEN UUO. THESE ROUTINES RETURN
        "TRUE" IF THEY SUCCEEDED, AND "FALSE" IF THEY
        FAILED.
%
%%

GLOBAL ROUTINE OPENIN(STATUS,LDEV,BUF)=
               (SKIP(OPEN(INCH,STATUS)));

GLOBAL ROUTINE OPENOUT(STATUS,LDEV,BUF)=
               (SKIP(OPEN(OUTCH,STATUS)));

%%
%

             THE PARAMETER PASSED TO LOOKUP OR ENTER MUST BE
        THE ADDRESS OF A FOUR-WORD CONTROL BLOCK, AS
        SPECIFIED IN THE MANUAL.  THE CONTENTS OF THIS
        CONTROL BLOCK WILL BE ALTERED BY THE UUO, AND
        CONSEQUENTLY MUST NOT RESIDE IN THE HISEG.  THESE
        ROUTINES RETURN "TRUE" IF THEY SUCCEED AND "FALSE"
        IF THEY FAIL.
```

```
%
%%

GLOBAL ROUTINE LOOKUP(LOOKUPBLOCK)=
                SKIP(LOOKUPUUO(INCH,LOOKUPBLOCK,0,1));

GLOBAL ROUTINE ENTER(ENTERBLOCK)=
                SKIP(ENTERUUO(OUTCH,ENTERBLOCK,0,1));

%%
%

             THE ROUTINE "READ" RETURNS ONE OF THE FOLLOWING
        VALUES:  -1 IF END-OF-FILE; -2 IF OTHER I/O ERROR; A
        POSITIVE NUMBER IS THE CHARACTER RETURNED.
%
%%

GLOBAL ROUTINE READ=
    BEGIN
        IF (COUNT(IBUFH)←.COUNT(IBUFH)-1) LEQ 0
          THEN
            IFSKIP IN(INCH) THEN RETURN
                IFSKIP STATZ(INCH,#740000)
                                THEN
                                        -1
                                ELSE
                                        -2;
          SCANI(PTR(IBUFH))
    END;

%%
%

             THE ROUTINE "WRITE" RETURNS "TRUE" IF IT
        SUCCESSFULLY WROTE OUT THE CHARACTER GIVEN AND
        "FALSE" IF IT DID NOT.
%
%%

GLOBAL ROUTINE WRITE(CHAR)=
    BEGIN
        IF (COUNT(OBUFH(←.COUNT(OBUFH)-1) LEQ 0
          THEN
            IFSKIP OUT(OUTCH)) THEN RETURN 0;
        REPLACEI(PTR(OBUFH),.CHAR);
        1
    END;

%%
%

             THE FOLLOWING ROUTINE PURGES THE OUTPUT BUFFER.
        THIS IS PARTICULARLY USEFUL FOR TTY OUTPUT.
%
%%
```

```
GLOBAL ROUTINE PURGEOUT=
        NOT SKIP(OUT(OUTCH));

%%
%

             THE FOLLOWING ROUTINES CLOSE THE CHANNELS AND
        RELEASE THE DEVICES.  THEIR VALUE IS UNDEFINED.
%
%%

GLOBAL ROUTINE CLOSEIN=(CLOSEUUO(INCH); RELEAS(INCH));

GLOBAL ROUTINE CLOSEOUT=(CLOSEUUO(OUTCH); RELEAS(OUTCH));

%%
%

             THIS ROUTINE IS USEFUL FOR OUTPUTTING MESSAGES
        TO THE TELETYPE.  IT TAKES ONE PARAMETER, THE
        POINTER TO AN ASCIZ STRING (E.G., PLIT ASCIZ
        'TEXT...').  ITS VALUE IS UNDEFINED.
%
%%

GLOBAL ROUTINE OUTMSG(TEXT)=
        TTCALL(3,TEXT,,1);


END ELUDOM;
```

Notes:

1. The MACHOP feature is used to provide the opcodes for the UUO's. Note that the names and values need not correspond to existing instruction names.

2. The SKIP macro is used to detect if the machine operation which is its argument skips upon return. Note the use of IFSKIP construct to detect this condition.

3. Note the use of SCANI and REPLACEI in the READ and WRITE routines, respectively.

4. Note the fact that the parameter lists to OPENIN and OPENOUT appear on the stack in the correct format for the OPEN UUO, which merely has to point to the first parameter of the list.

5. Indirect addressing is used in the LOOKUP, ENTER and OUTMSG routines. This produces slightly better code.

6. The ENTRIES declaration is used in the module head to make the routine names available for library search by the loader.

7. The macros defining the count and pointer fields are used to make the code read clearly. Thus the essence of the operations is seen without the details.

Example 5: A Sample Program Using Example 4

This is a somewhat trivial example of a program which transfers an ASCII file from the disk to the printer. We will assume in this example that the name of the file is "FILE.EXT"". Note that no test is made for the existence of the file. If it does not exist (i.e., LOOKUP fails) this will appear as an input error to READ.

```
MODULE DSKLPT(STACK)=
BEGIN
%%
%

                THIS MODULE TRANSFERS ASCII FILES TO THE
        PRINTER.  FOR ILLUSTRATIVE PURPOSES HERE, THE INPUT
        FILE IS ALWAYS CALLED "FILE.EXT".
%
%%

EXTERNAL
        CLOSEIN, CLOSEOUT,
        OPENIN, OPENOUT, LOOKUP, OUTMSG, READ, WRITE;

OWN
        LBLOCK[4],
        T;

GLOBAL
        OBUFH[3], IBUFH[3];

MACRO
        RESET=CALLI(0,0)$,
        STOP=CALLI(1,#12)$;

MACHOP
        CALLI=#047;
LABEL TRANSFERS;
%%
%
        HERE IS THE MAIN PROGRAM
%
%%

        RESET;

UNTIL OPENOUT(1, SIXBIT 'LPT', OBUFH↑18) DO
        (OUTMSG(PLIT ASCIZ '?? NO LPT?M?J'); STOP);

UNTIL OPENIN(1, SIXBIT 'DSK', IBUFH<0,0>) DO
        (OUTMSG(PLIT ASCIZ '?? NO DSK?M?J'); STOP);


        LBLOCK[0]←SIXBIT 'FILE';
        LBLOCK[1]←SIXBIT 'EXT';
        LBLOCK[2]←LBLOCK[3]←0;
        LOOKUP(LBLOCK<0,0>);

        TRANSFER: WHILE (T←READ()) GEQ 0 DO
                (IF .T NEQ 0 THEN IF NOT WRITE(.T)
                             THEN LEAVE TRANSFER
                                  WITH(T←-3));
        CLOSEIN(); CLOSEOUT();

        CASE .T+3 OF
                SET
                % -3 % OUTMSG(PLIT ASCIZ '??LPT ERROR?M?J');
                % -2 % OUTMSG(PLIT ASCIZ '??DSK ERROR?M?J');
                % -1 % OUTMSG(PLIT ASCIZ 'DONE?M?J');
                TES;

END;
```

Notes:

1. The STACK declaration is used because this is the main program. Since no parameters are given, the stack is an OWN array, size 1000 words, with the name STACK.

2. Macros define the CALLI UUO's for RESET and EXIT ("STOP").

3. The STOP macro is used in loops, which allows the user to issue an ASSIGN command and type CONT to resume if an error occurs.

4. The LEAVE expression is used to terminate processing in the case of an output error. The loop condition is used to terminate processing in the case of an input error.

5. The CASE expression is used to output the appropriate message. Note the use of comments to show which value of T will execute a given expression within the CASE. This method of commenting CASE expressions makes them very readable.

6. The attributes SIXBIT and ASCIZ are used to define strings.

7. The escape convention (? character) is used in the messages to obtain control characters. Note the use of the double question mark(??) to obtain a single question mark.

Example 6: Monitor I/O

These routines supply low-level support for programs which run in the user mode under the DECsystem-10 timesharing monitor. Although not all I/O facilities are shown in this example, the same methods may be used to provide any I/O facility in BLISS-10.

These routines differ from those in Example 4 in that they allow the channel number to be supplied as one parameter to each routine. There is no restriction to doing input or output on only one channel. These routines use about 128 words.

```
MODULE IO(ENTRIES=(LOOKUP,ENTER,OPEN,CLOSE,
                PURGEOUT,OUTMSG,READ,WRITE))=

BEGIN
%%
%

             THIS MODULE PROVIDES SOME I/O EXAMPLES USING
        BLISS-10.  FOR THIS SET OF EXAMPLES, WE WILL USE
        DYNAMIC CHANNEL ASSIGNMENT.
%
%%

%%
%
        HERE ARE SOME USEFUL MACHINE OPERATIONS
%
%%

MACHOP
        CALLI=#047,
        TTCALL=#051,
        XCT=#256;

BIND
        OPENUUO=#050,
        IN=#056,
        OUT=#057,
        GETSTS=#062,
        STATZ=#063,
        CLOSEUUO=#070,
        RELEAS=#071,
        LOOKUPUUO=#076,
        ENTERUUO=#077;

%%
%

             THE FOLLOWING MACRO RETURNS "TRUE" IF THE
        EXPRESSION GIVEN AS ITS PARAMETER SKIPS, AND FALSE
        IF IT DOES NOT.
%
%%

MACRO
        SKIP(OP)=IFSKIP (OP) THEN 1 ELSE 0$;

%%
%
        HERE ARE SOME USEFUL MACROS
%
%%

MACRO
        RESET=CALLI(0)$,

        ICOUNT(CHNL)=(.BUFH[CHNL]<0,18>+2)<0,36>$,
        IPTR(CHNL)=(.BUFH[CHNL]<0,18>+1)<0,36>$,
        OCOUNT(CHNL)=(.BUFH[CHNL]<18,18>+2)<0,37>$,
        OPTR(CHNL)=(.BUFH[CHNL]<18,18>+1)<0,36>$;
%%
%

             THIS VECTOR KEEPS THE BUFFER HEADER POINTERS
        PASSED TO "OPEN".
%
%%

OWN
        BUFH[16];
%%
%

             THE FOLLOWING MACROS ARE USED TO CONSTRUCT
        INSTRUCTIONS AND EXECUTE THEM.
%
%%

MACRO
        MAKEOP(OP,REG,ADDR)=(OP)<0,0>↑27+(REG)<0,0>↑23+(ADDR)<0.0>$,
        EXECUTE(X)=(REGISTER Q; Q←X;SKIP(XCT(0,Q)))$,
        IND=0,0,0,1$;

%%
%

             "OPEN" TAKES FOUR PARAMETERS:  THE CHANNEL
        NUMBER, THE DEVICE STATUS (INCLUDING DATA MODE), THE
        LOGICAL DEVICE NAME, AND THE BUFFER POINTERS, THE
        LATTER THREE PRECISELY AS SPECIFIED FOR THE OPEN
        UUO.  THIS ROUTINE RETURNS "TRUE" IF IT SUCCEEDED,
        AND "FALSE" IF IT FAILED.
%
%%

GLOBAL ROUTINE OPEN(CHNL,STATUS,LDEV,BUF)=
        (BUFH[.CHNL]←.BUF;
         EXECUTE(MAKEOP(OPENUUO,.CHNL,STATUS)));

%%
%

             THE SECOND PARAMETER PASSED TO LOOKUP OR ENTER
        MUST BE THE ADDRESS OF FOUR-WORD CONTROL BLOCK, AS
        SPECIFIED IN THE MANUAL.  THE CONTENTS OF THIS
        CONTROL BLOCK WILL BE ALTERED BY THE UUO, AND
        CONSEQUENTLY MUST NOT RESIDE IN THE HISEG.  THESE
        ROUTINES RETURN "TRUE" IF THEY SUCCEED AND "FALSE"
        IF THEY FAIL.
%
%%

GLOBAL ROUTINE LOOKUP(CHNL,LOOKUPBLOCK)=
        EXECUTE(MAKEOP(LOOKUPUUO,.CHNL,LOOKUPBLOCK<IND>));
```

```
GLOBAL ROUTINE ENTER(CHNL,ENTERBLOCK)=
        EXECUTE(MAKEOP(ENTERUUO,.CHNL,ENTERBLOCK<IND>));

%%
%

             THE ROUTINE "READ" RETURNS ONE OF THE FOLLOWING
        VALUES:  -1 IF END-OF-FILE; -2 IF OTHER I/O ERROR; A
        POSITIVE NUMBER IS THE CHARACTER RETURNED.
%
%%

GLOBAL ROUTINE READ(CHNL)=
    BEGIN
        IF (ICOUNT(.CHNL)←.ICOUNT(.CHNL)-1) LEQ 0
          THEN
            BEGIN
                IF EXECUTE(MAKEOP(IN,.CHNL,0))
                 THEN
                  RETURN
                  (IF EXECUTE(MAKEOP(STATZ,.CHNL,#740000))
                        THEN
                                -1
                        ELSE
                                -2)
            END;
        SCANI(IPTR(.CHNL))
    END;

%%
%

             THE ROUTINE "WRITE" RETURNS "TRUE" IF IT
        SUCCESSFULLY WROTE OUT THE CHARACTER GIVEN AND
        "FALSE" IF IT DID NOT.
%
%%

GLOBAL ROUTINE WRITE(CHNL,CHAR)=
    BEGIN
        IF (OCOUNT(.CHNL)←.OCOUNT(.CHNL)-1) LEQ 0
          THEN
            (IF EXECUTE(MAKEOP(OUT,.CHNL,0)) THEN RETURN 0);
        REPLACEI(OPTR(.CHNL),.CHAR);
        1
    END;

%%
%

             THIS ROUTINE PURGES THE OUTPUT BUFFER.  THIS IS
        PARTICULARLY USEFUL FOR TTY OUTPUT.
%
%%

GLOBAL ROUTINE PURGEOUT(CHNL)=
        NOT EXECUTE(MAKEOP(OUT,.CHNL,0));
```

```
%%
%

          THE FOLLOWING ROUTINE CLOSES THE CHANNEL AND
      RELEASES THE DEVICE.  ITS VALUE IS UNDEFINED.
%
%%

GLOBAL ROUTINE CLOSE(CHNL)=
        (EXECUTE(MAKEOP(CLOSEUUO,.CHNL,0));
         EXECUTE(MAKEOP(RELEAS,.CHNL,0)));

%%
%

          THIS ROUTINE IS USEFUL FOR OUTPUTTING MESSAGES
      TO THE TELETYPE.  IT TAKES ONE PARAMETER, THE
      POINTER TO AN ASCIZ STRING (E.G., PLIT ASCIZ
      'TEXT...').  ITS VALUE IS UNDEFINED.
%
%%

GLOBAL ROUTINE OUTMSG(TEXT)=
        TTCALL(3,TEXT,,1);


END ELUDOM;
```

Notes:

1. The MACHOP feature is used to provide the opcodes for the UUO's CALLI and TTCALL, as well as the machine operation XCT.

2. The BIND declaration is used to associate names with the values of the UUO opcodes. The reason this is done, instead of declaring these names as MACHOPs (as is done in example 4) will be discussed below.

3. The SKIP macro is used to detect whether the machine operation which is its argument skips upon return. Note the use of the IFSKIP construct.

4. In any UUO dealing with I/O, the register field is the channel number. In order to provide for varying channel numbers, the value of this field must be evaluated at this time. However, the BLISS-10 compiler requires that this field evaluate to a constant at compile time. In order to allow this field to vary at execution time, we construct the instruction and execute it by means of the XCT instruction. The macro MAKEOP constructs an instruction, given the parameters of its opcode, register field, and address field. The macro EXECUTE takes the expression given as its parameter, considers it an instruction, and executes it. Note that EXECUTE assumes the possibility that the instruction will skip upon return, and its value is the value of the SKIP macro. A minor restriction is that the name"Q" may not be used in the argument to the EXECUTE macro.

However, the choice of this name is arbitrary and any name may be used in coding the macro.

5. Note the use of SCANI and REPLACEI in the READ and WRITE routines, respectively.

6. Note the fact that the parameter list to OPEN appears on the stack in the correct format for the OPEN UUO, which merely has to point to the first parameter of the list.

7. Indirect addressing is used in the LOOKUP, ENTER and OUTMSG routines. This produces slightly better code.

8. The ENTRIES declaration is used in the module head to make the routine names available for library search by the loader.

9. The macros defining the count and pointer fields are used to make the code read clearly. The essence of the operations is seen without the details.

Example 7: A Sample Program Using Example 6

This is a somewhat trivial example of a program which transfers an ASCII file from the disk to the printer. We will assume in this example that the name of the file is "FILE.EXT". This program is basically similar to the program in example 5.

```
MODULE DSKLPT(STACK)=
BEGIN
%%
%

             THIS MODULE TRANSFERS ASCII FILES TO THE
        PRINTER.  FOR ILLUSTRATIVE PURPOSES HERE, THE INPUT
        FILE IS ALWAYS CALLED "FILE.EXT".
%
%%

EXTERNAL
        CLOSE, OPEN, LOOKUP, OUTMSG, READ, WRITE;

BIND
        INCH=1,                      % INPUT CHANNEL NUMBER %
        OUCH=2;                      % OUTPUT CHANNEL NUMBER %

OWN
        LBLOCK[4],
        T,
        OBUFH[3], IBUFH[3];

MACRO
        MSG(X)=OUTMSG(PLIT ASCIZ X)$,
        RESET=CALLI(0,0)$,
        HALT=CALLI(0,#12)$,
        STOP=CALLI(1,#12)$;
```

```
MACHOP
         CALLI=#047;
LABEL TRANSFER;
%%
%
         HERE IS THE MAIN PROGRAM
%
%%

         RESET;

UNTIL OPEN(OUCH,1, SIXBIT 'LPT', OBUFH↑18) DO
         (MSG('?? NO LPT?M?J'); STOP);

UNTIL OPEN(INCH,1, SIXBIT 'DSK', IBUFH<0,0>) DO
         (MSG('?? NO DSK?M?J'); STOP);


         LBLOCK[0]←SIXBIT 'FILE';
         LBLOCK[1]←SIXBIT 'EXT';
         LBLOCK[2]←LBLOCK[3]←0;

         IF NOT LOOKUP(INCH,LBLOCK<0,0>)
                 THEN
                 (MSG('??FILE.EXT NOT FOUND?M?J');
                 HALT);

               TRANSFER:         WHILE (T←READ(INCH)) GEQ 0 DO
                 (IF .T NEQ 0 THEN
                       IF NOT WRITE(OUCH,.T)
                              THEN LEAVE TRANSFER WITH (T←-3));

         CLOSE(INCH); CLOSE(OUCH);

         CASE .T+3 OF
                 SET
                 % -3 % MSG('??LPT ERROR?M?J');
                 % -2 % MSG('??DSK ERROR?M?J');
                 % -1 % MSG('DONE?M?J');
                 TES;
END;
```

Notes:

1. The STACK declaration is used because this is the main program. Since no parameters are given, the stack is an OWN array, size 1000 words, with the name STACK.

2. Macros define the CALLI UUO's for RESET and EXIT ("STOP" and "HALT"). Note that the EXIT UUO is given two different names, depending on whether its accumulator field is zero or nonzero (see the DECsystem-10 monitor manual for the significance of this).

3. The STOP macro is used in the loops, which allows the user to issue an ASSIGN command and type CONT to resume if an error occurs.

4. The LEAVE expression is used to terminate processing in the case of an output error. The loop condition is used to terminate processing in the case of an input error.

5. The CASE expression is used to output the appropriate message after the loop is terminated. Note the use of comments to show which value of T will execute a given expression within the CASE. This method of commenting CASE expressions makes them very readable.

6. The attributes SIXBIT and ASCIZ are used to define strings.

7. The escape convention (? character) is used in the messages to obtain control characters. Note the use of the double question mark (??) to obtain a single question mark.

8. Note the use of the MSG macro, which reduces a great deal of clutter in coding the messages. Compare the messages in this example with those of example 5.

APPENDIX A
BLISS-10 SYNTAX

```
module              ::=block/
                       module-head block/
                       module-head block ELUDOM/block ELUDOM
module-head            modulename (mdle-parms)=
mdle-parms          ::=mdle-parm,mdle-parm,mdle-parms
mdle-parm           ::=lst-parm/err-parm/opt-parm/
                       mach-list-parm/hi-seg/
                       inspct-parm/syntx/dregs/-resrv
                       expnd-parm/reg-save-parm/
                       lo-seg-parm/glbl-rtn-parm/stack-parm
                       entry-parm/timer-parm/timing-parm
                       ccl-parm/sreg/vreg/breg/freg
                    fsv-parm/xrf-parm/eng-parm/
                    strt/prlog/headf/deb/parm
                    vnum
lst-parm            ::=LIST/NOLIST
err-parm            ::=ERRS/NOERRS
opt-parm            ::=OPTIMIZE/NOOPTIMIZE
mach-lst-parm       ::=MLIST/NOMLIST
hi-seg              ::=HISEG
inspct-parm         ::=INSPECT/NOINSPECT
syntx               ::=SYNTAX
dregs               ::=DREGS=E
resrv               ::=RESERVE(Ei.,....En)
expnd-parm          ::=EXPAND/NOEXPAND
reg-save-parm       ::=NORSAVE/RSAVE
lo-seg-parm         ::=LOSEG
glbl-rtn-parm       ::=GLOROUTINES/NOGLOROUTINES
entry-parm          ::=ENTRIES=(ni,...nm)
timer-parm          ::=TIMER
timing-parm         ::=TIMING/NOTIMING
ccl-parm            ::=CCL
sreg                ::=SREG=E
vreg                ::=VREG=E
freg                ::=FREG=E
fsv-parm            ::=FSAVE/NOFSAVE
xrf-parm            ::=XREF/NOXREF
eng-parm            ::=ENGLISH/NOENGLISH
strt                ::=START
prlog               ::=PROLOG
headf               ::=HEADFILE file-spec
deb-parm            ::=DEBUG/NODEBUG
vnum                ::=VERSION=VNO
vno                 ::=<major><minor>(<edit>)-<who>
major               ::=octal in range 0-777
minor               ::=letter A-Z
edit                ::=octal in range 0-777777
who                 ::=oit
block               ::=BEGIN blockbody END/
                        (blockbody)
blockbody           ::=decls exprs
decls               ::=decl/decls;decl
cmpdn-exprs         ::=BEGIN exprs END/(exprs)
exprs               ::=E/label:E/E;exprs
                        E semicolon exprs/empty
```

```
comment           ::=restofline end-in-sym/%strng-no-pct%/empty
E                 ::=smpl-expr/cntrl-expr
smpl-expr         ::=P11=E/P11
P11               ::=P10/P11 XOR P10/P11 EQV P10
P10               ::=P9/P10 OR P9
P9                ::=P8/P9 AND P8
P8                ::=P7/NOT P7
P7                ::=P6/P6 rel-op P6
p6                ::=P5/-P5/P6+P5/P6-P5
P5                ::=P4/P5*P4/P5/P4/P5 MOD P4
P4                ::=P3/P4↑P3
P3                ::=P2/.P3
P2                ::=P1/P1<pntr-parms>
P1                ::=literal/name/E[expr-lst]/
                     P1(expr-lst)/P1()/block/
                     cmpnd-expr
pntr-parms        ::=pos,size,indx,indrct
rel-op            ::=EQL/NEQ/LSS/LEQ/GTR/GEQ/
                     EQLU/NEQU/LSSU/GTRU/GEQU
literal           ::=number/string/plit
string            ::=string-type quoted-string
strng-type        ::=ASCII/ASCIZ/RADIX50/SIXBIT/empty
quoted-string     ::= left adjusting/right adjusting
left-adj-string   ::= 'string'
right-adj-string  ::= "string"
number            ::=decimal/octal
decimal           ::=digit/decimal digit
floating          ::=decimal, decimal/
                     decimal,decimal exponent/
exponent          ::=E decimal/E+decimal/E-decimal(1)
octal             ::=#oit/octal oit
oit               ::=0/1/2/3/4/5/6/7
digit             ::=0/1/2/3/4/5/6/7/8/9
plit              ::=PLIT plitarg
plitarg           ::=load-time-expr/long-string/triple
triple            ::=(triple-item-lst)
triple-item-lst   ::=triple-item/triple-item,triple-item-lst
triple-item       ::=load-time-expr/long-string/
                     dup-fctr:plitarg
dup-fctr          ::=cmpl-time-expr
name              ::=letter/name letter/name digit
letter            ::=A/A/B/C/.../Z/a/b/c.../z
plit-name-bind    ::=name names/
                     name globally names/
                     name indexes/
                     name globally indexes
cntrl-expr        ::=cndtl-expr/loop-expr/choice-expr/
                     escp-expr/co-rtn-expr
cndtl-expr        ::=IF E1 THEN E2 ELSE E3/
                     IF E1 THEN E2
                  IFSKIP E1 THEN E2 ELSE E3/
                  IFSKIP E1 THEN E2
```

----

(1) E in this case is the letter E rather than an expression.

```
loop-expr         ::=WHILE E1 DO E2/
                     UNTIL E1 DO E2/
                     DO E1 WHILE E2/
                     DO E1 UNTIL E2
                     INCR name FROM E1 TO E2 BY E3 DO E4/
                     DECR name FROM E1 TO E2 BY E3 DO E4
esc-expr          ::=LEAVE label WITH E
                     EXITLOOP E
                     RETURN E
                     LEAVE label
                     SIGNAL E
escpe-expr        ::=envt-lvl escp-val/RETURN escp-val
envt              ::=EXIT/EXITBLOCK/EXITCOMPOUND/EXITLOOP/EXITCOND
                     EXITCASE/EXITSET/EXITSELECT/BREAK
                     /EXITCOMP/EXITCONDIT
lvl               ::=E/empty
escp-val          ::=E/empty
choice-expr       ::=CASE E OF SET expr-set TES/
                     SELECT E OF NSET a-expr-set TESN
expr-set          ::=E/;expr-set/E;expr-set/empty
a-expr-set        ::=a-E-E;a-expr-set/empty
a-E               ::=E:E/OTHERWISE:ELWAYS:E

decl              ::=routn-decl/
                     fetn-decl/
                     strc-decl/
                     macro-decl/
                     alloc-dec/
                     map-decl/
                     labl-decl/
                     un-decl/
                     ext-fwd-decl/
                     bind-decl/
                     swtch-decl/
                     reqre-decl/
alloc-decl        ::=alloc-typ msid-lst
alloc-typ         ::=OWN/LOCAL/GLOBAL/EXTERNAL/REGISTER
msid-lst          ::=msid-elmt/msid-elmt,msid-lst
msid-elmt         ::=strc sized-chnks
strc              ::=strc-name/empty
sized-chnks       ::=size-chnk/size-chnk,sized-chnks
size-chnk         ::=id-chnk/id-chnk[expr-lst]
id-chnk           ::=name/name:id-chnk

strc-decl         ::=STRUCTURE name strc-frml-lst=strc-size E
strc-frml-lst     ::=[name-lst]/empty
strc-size         ::=[E]/empty
fctn-decl         ::=FUNCTION name(name-lst)-E/
                     FUNCTION name=E/
                     ROUTINE name(name-lst)=E/
                     ROUTINE name=E/
                     GLOBAL ROUTINE name(name-lst)=E/
                     GLOBAL ROUTINE name=E
fctn-call         ::=fctn-expr(exp-lst)/fctn-expr()
fctn-expr         ::=literal/name/cmpnd-expr/block/
                     name[expr-list]
expr-lst          ::=E/ expr-lst,E
ext-fwd-decl      ::=EXTERNAL  name-par-lst/
```

```
                 FORWARD  name-par-lst
name-par-lst    ::=name-par/name-par-list,name-par
name-parm       ::=name(E)/name

mac-decl        ::=MACRO dfn-lst
dfn-lst         ::=dfn/dfn-lst,dfn
dfn             ::=name(name-lst)=strng-no-$ $/
                   name=strn-no-$ $

map-decl        ::=MAP msid-lst/MAP link-name msid-lst
bind-decl       ::=BIND equ-lst/GLOBAL BIND equ-lst
equ-lst         ::=equ/equ, equ-lst
equ             ::=msid-elmt=E
equ-lst         ::=equ/equ,equ-lst
equ             ::=msid-elmt=E

label-decl      ::=LABEL labl-lst
labl-lst        ::=name/name,labl-lst

undecl-         ::=UNDECLARE name-lst

swtch-decl      ::=SWITCHES swtch-lst
swtch-lst       ::=swtch/swtch-lst
swtch           ::=lst-parm/err-parm/mach-lst-parm/inspct parm/
                   opt-parm/expnd-parm/glbl-rtn-parm/reg-save-parm
                   lo-seg-parm/timing-parm
lst-parm        ::=LIST/NOLIST
err-parm        ::=NOERS
mach-lst-parm   ::=MLIST/NOMLIST
inspct-parm     ::=INSPECT/NOINSPECT
opt-parm        ::=OPTIMIZE/NOOPTIMIZE
expnd-parm      ::=EXPAND/NOEXPAND
GLBL-RTN-PARM   ::=GLOROUTINES/NOGLOROUTINES
reg-save-parm   ::=NORSAVE/RSAVE
lo-seg-parm     ::=LOSEG
timing-parm     ::=TIMING/NOTIMING
reqre-decl      ::=REQUIRE file-spec
file-spec       ::=file-name-spec/device-file-name-spec
device          ::=name
file-name-spec  ::=file-name/file-name[ppn-spec]
file-name       ::=name/name-name/empty
ppn-spec        ::=DECppn
DECppn          ::=octal, octal
```

APPENDIX B
DESCRIPTION OF NON-TERMINALS OF BLISS-10

The following list of non-terminals attempts to describe text for the mnemonic uses of the syntax of BLISS-10 constructs. The non-terminal mnemonics appear in an order which corresponds to the definition of the construct in the specification proper.

module:
: An independently compliable and linkable entity.

module-head
: The beginning of a module consisting of a module name and parameters which affect the type of module produced.

mdle-parms
: module-parameters

mdle-parm
: A module-parameter, taken from the set LIST, NOLIST, NOERS, OPTIMIZE, NOOPTIMIZE, MLIST, NOMLIST, HISEG, INSPECT, NOINSPECT, SYNTAX, DREGS=e, RESERVE(ei,...en), EXPAND, NOEXPAND, SREG=e, VREG=e, FREG=e, NORSAVE, RSAVE, LOSEG, STACK, GLOROUTINES, NOGLOROUTINES, ENTRIES=(ni,...nm), TIMER, TIMING, NOTIMING,CCL.

block
: Declarations and expressions, or compound expressions bracketed by either BEGIN-END or ().

blockbody
: Declarations and expressions or compound expressions.

decls
: Declarations; declarations establish relationships, allocate storage, and define data structures of a program.

cmpnd-exprs
: Compound-expressions; a compound-expression consists of a string of expressions separated by semicolons <;> or semicolon and enclosed in a BEGIN-END pair or (). Note compound-expressions differ from a block in that they do not contain declarations.

exprs
: Expressions; Atoms connected by operators possibly labeled, possibly strung together, and separated by semi-colons. Expressions become compound-expressions when enclosed in BEGIN-END or ().

E
: Expressions: either a string of atoms connected by operators with no intervening punctuations, or a control expression.

smpl-expr
: Simple-expression; A string of primaries (language elements reduceable to a single value) connected by operators with no intervening punctuation.

Pn where n has some integer value.
Primary; An irreduceable language element which computes to a single value. These consist of literals, names, structure names, routine calls, blocks, and compound-expressions.

rel-op
Relational operators.

Comment
A string enclosed by a percent symbols (%...%) or a string which begins with an exclamation point (!) and terminates with a carriage return.

restofline
The string following an exclamation point terminated by the end-of-line-symbol.

end-ln-sym
End-of-line-symbol; The character or characters which constitute line termination, (currently a carriage return; line feeds are ignored).

strng-no-pct
String-with-no-percent; A string of characters which does not contain a percent.

literal
An atom which represents itself at compile time.

decimal
A decimal integer consisting of any number of the digits 0 thru 9.

floating
A floating point number in either decimal or exponential format.

exponent
A floating point exponent in the form E+decimal or E-decimal.

octal
An octal integer consisting of any number of the digits 0 thru 7.

oit
Octal digit.

digit
Decimal digit.

quoted-string
A string of any length enclosed in quotes.

left-adj-string
A string enclosed in single quotes which is left adjusted in a word.

right-adj-string
A string enclosed in double quotes which is right adjusted in a word.

plit
    A pointer to a literal.

plitarg
    The arguments of a PLIT.

load-time-expr
    A load-time-expression; An expression whose value must be known at the completion of the loading process. Can serve as a PLIT argument.

long-string
    A string enclosed in quotes which has a length greater than one word (7 characters for SIXBIT ; 6 characters for ASCII; 5 characters for ASCIZ). Can serve as a PLIT argument.

triple
    An argument type to a PLIT enclosed in parentheses ().

triple-item-lst
    One or more PLIT arguments separated by commas.

dup-fctr
    Duplication factor; A repeat count applicable to any of argument types of a PLIT.

cmple-time-expr
    Compile-time-expression; An expression whose value is determined at the completion of a compilation.

cntrl-expr
    Control-expression; expressions which may alter the standard sequential execution sequence within BLISS-10 programs.

cndtl-expr
    Conditional-expression

loop-expr
    Loop-expression

esc-expr
    Escape-expression; Expression which enables the ordinary termination of one context and the entry to another.

choice-expr
    Choice-expression

expr-set
    Expression set; The argument list of a CASE expression.

expr-lst
    A list or expressions separated by commas.

a-expr-set
    Associative-expression-set; the argument list for a SELECT expression.

a-E	Associative-expression
decl	Declaration
alloc-decl	Allocation-declaration; establishes names for storage segments; specifies their type and size.
alloc-typ	Allocation-type; a specification of the type of storage the compiler should set aside for units of the specified structure.
msid-lst	Main-structure-identifier-list; specifies names and sizes for structures.
strc-name	structure-name; a name associated with a structure declaration.
strc-decl	Structure-declaration; establish a data structure via size specifications and an accessing algorithm.
strc-frml-lst	Structure-formal-list; a list of the formal parameters used in the structure-size (strc-size) specification and the accessing algorithm for the structure.
fctn-decl	Routine-declaration; define a function or routine.
glbl-decl	Global-declaration; specifies a routine name as GLOBAL.
ext-fwd-decl	External, or forward declaration; provides a mechanism for specifying the number of arguments required by a routine either defined externally, or a routine defined such that its body contains a call on a second routine not yet declared.
name-parm-lst	Name-parameter-list; list of name-parms acceptable in an EXTERNAL or FORWARD declaration.
name-parm	Name-parameter; name or a routine which may or may not carry a specification of the number of parameters expected by a routine call.
macro-decl	Macro-declaration; define a macro
dfn-lst	Definition-list; an argument list in a macro definition

fxd-parms
    Fixed-parameters;

map-decl
    Map-declaration; redefine access algorithm for a storage segment.

bind-decl
    Bind-declaration; establishes a name-value correspondence the value being established at block entry.

equ-lst
    Equivalence-list; list of atoms permitted in a BIND declaration.

labl-decl
    Label-declaration; establish a label for use in a LEAVE expression.

labl-lst
    Label-list; a list of names which the compiler will interpret as labels.

un-decl
    Un-declare-declaration; render names undefined within a block.

swtch-decl
    Switch-declaration; provide compilation control at the block level.

swtch-lst
    Switch-lst; a list of arguments valid in a switch-declaration

reqre-decl
    Require declaration; enables the programmer to include a previously stored text file in his source during compilation.

file-spec
    File specification; a pointer to a file where the text to be included can be found.

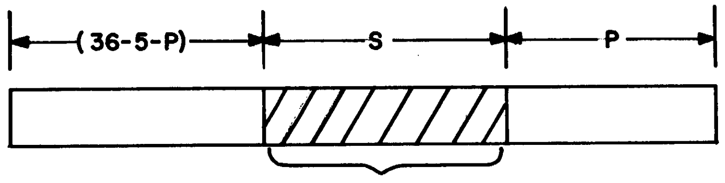
APPENDIX C
BLIS10 RESERVED WORDS

The following list contains all reserved words which appear in BLISS-10 version 4:

ABS	ELSE	GEQ	OF	WHILE
ALLMACHOP	ELUDOM	GLOBAL	OFFSET	WITH
ALWAYS	END	GLOBALLY	OR	
AND	EQL	GTR	OTHERWISE	XOR
AT	EQV		OWN	
	EXCHJ	IF		
		IFSKIP		
BEGIN	EXIT	INCP	PLIT	
BIND	EXITCASE	INCR		
BREAK	EXITCOMP	INDEXES	REGISTER	
BY	EXITCOMPOUND		REPLACEI	
	EXITCOND	LABEL	REPLACEN	
			REQUIRE	
CASE	EXITCONDIT	LEAVE	RETURN	
COPYII	EXITLOOP	LENGTH	ROUTINE	
COPYIN	EXITSELECT	LEQ		
COPYNI	EXITSET	LOCAL	SCANI	
COPYNN	EXTERNAL	LSS	SCANN	
CREATE			SELECT	
	FIRSTONE	MACHOP	SEMICOLON	
	FIX	MACRO	SET	
DEBUG	FLOAT	MAP	SIGN	
DECR	FORWARD	MOD	STRUCTURE	
DIV	FROM	MODULE	SWITCHES	
DO	FUNCTION			
		NAMES	TES	
		NEQ	TESN	
		NOT	THEN	
		NSET	TO	
			TRAP	
			UNDECLARE	
			UNTIL	

APPENDIX D
WORD FORMATS

<P,S> refers to a field S bits wide and P bits up from the right hand end of the word, thus:



referenced partial word

The format of a pointer is

P = <30,6>	Position
S = <24,6>	Size
I = <22,1>	Indirect address
X = <18,4>	Index
Y = <0,18>	Word address

The format of a non-I/O instruction is

F = <27,9>	Function code
A = <23,4>	Accumulator
I,X,Y as above	

The format of an integer number is

```
SIGN      = <35,1>
MAGNITUDE = <0,35>
```

The format of a floating point number is

```
SIGN     = <35,1>
EXPONENT = <27,8>
MANTISSA = <0,27>
```

APPENDIX E
BLISS-10 ERROR MESSAGES

Error Message Notes

The following is a complete list of compiler error numbers with explanations. Unless otherwise noted, an error is a fatal syntax error.

* indicates that this is a warning message and code generation, if requested, will continue.

indicates that the compiler itself has made an error and will attempt to reinitialize itself. It is recommended that a new core image of the compiler be obtained before another compilation is attempted. Please report these errors via SPR.

The majority of the errors are fatal syntax errors. Code generation is terminated on any of these conditions.

NUMBER	MESSAGE
0*	Undeclared identifier.
1	Error in simple expression.
2	Not the correct matching close bracket.
3	Expressions must be separated by a delimiter.
4	An operator must be followed by a simple expression.
5	A relational expression must not be followed by a relational operator.
6	A unary (binary) operator must (not) be preceded by a delimiter.
7	A control expression must not be used as a subexpression.
10	Left part of an assignment is incorrect.
11	Too many ←'s (current implementation allows 8).
12	Right hand side of an assignment is incorrect.
13	An actual parameter expression should not be empty.
14	A simple expression should be followed by a delimiter.
15	A subscript expression should not be empty.
16	Too many subscripts (current implementation allows 8).

17	A null selector in CASE expression is illegal.
20	OF must be followed by SET in CASE statement.
21	Incorrect escape expression.
22	Missing control variable in INCR or DECR.
23	The constituent expressions of a complex expression should not be empty.
25	Declarations are only allowed in a block head.
30	Current close bracket does not match marked open bracket (paired with error 31). This pointer marks the open parenthesis.
31	This pointer marks the incorrect close parenthesis (paired with error 30).
36	Illegal control variable name in INCR or DECR.
37	Empty condition in WHILE-DO, UNTIL-DO, DO-WHILE, or DO-UNTIL.
40	Illegal up level addressing. Specifically, when the compiler sees a reference to an identifier of type local, formal, function, structure formal, or a bind which is not a compile time constant expresion, it tests to insure that the identifiers have not been declared at a level outside (lower than) that of the latest routine declaration. Since all the above types are referenced via the stack, displays are required in order to keep track of the stack reference pointer at outer levels. Since the displays are not available within a routine, it is impossible to reference specific stack positions which are set aside in outer blocks.
41	Too many parameters in a pointer expression.
42	Too many close brackets, or not enough open brackets (compiler exited to highest level before the EOF on input file).
43*	As 42, except warning only, recovery attempted.
44	FROM-TO-BY-DO out of order in INCR/DECR expression.
45	Empty DO part, may not be defaulted in INCR/DECR expression.
46	Empty condition in IF-THEN-ELSE not permitted.
47	Missing THEN.
50	Empty FROM, TO, or BY expression in INCR/DECR.
51	Number of levels in escape expression is not a literal.

52	Missing ']' in number levels part of an escape expression.
53	Empty expression not permitted as pointer-pointer in special function.
54	Missing ')' in a special function.
55	Missing 'OF' in select expression.
56	Missing or misplaced 'NSET' in SELECT expression.
57	Labeling expression of nset-element may not be empty.
60	Missing or misplaced ":" in a SELECT expression.
61	Missing or misplaced TESN in SELECT expression.
62	Empty list element in SELECT expression.
63	'SET' is not an allowed expression beginner.
64	No '(' after EXCHJ.
65	Empty new-base expression in EXCHJ.
66	Missing ')' in EXCHJ.
67	Missing AT in CREATE.
70	Missing AT-expression or LENGTH in CREATE.
71	Missing LENGTH-expression or THEN in CREATE.
72	Missing '(' after CREATE.
74	Symbol to be declared is not an identifier.
75	Missing "=" on a routine, function, or structure declaration.
76	Missing formal parameter list right delimiter, i.e., ")""]"",".
77	Missing right bracket on the size portion of a "namesize".
100	Missing delimeter on a list, i.e., "," or ";".
101	Missing ")" on a name par.
103	Missing "=" in a MACHOP declaration.
104	Missing "," ":" ";" in allocation declaration.
105*	An idenfifier precedes a declaration. The identifier is ignored.
106*	Structure access not to an identifier, e.g., l[e], 'vector' assumed.

126*	Register is neither reserved nor 'system' type, see MODULE declaration.
127*	Register value out of range (0-15).
130	Register number is not a literal.
131*	Attempted structure access to a variable which has not been mapped, 'vector' assumed.
132*	Extra incarnation actuals...ignored.
133	Size expression must not be a block.
134	Symbol may not be addressed, and hence may not be mapped.
135	Invalid expression in a FORWARD declaration.
136	Invalid expression in a MACHOP declaration.
137	May not map a symbol of this type.
140	Attempting to map onto an undeclared structure.
141	Incarnation actual or resulting size expression is not a literal.
142	Delimiters for this balanced string of macro arguments do not match.
143	Symbol previously declared in the current context (blocklevel).
144	Invalid attempt to escape from routine or function.
145*	Warning: Using a temporary register may invalidate code.
146	Register position of a machine operation must be register name or literal.
147	The indirect field in a machine language statement must be a compile time constant.
150	A machine language expression involving a name declared as a MACHOP is missing its close parenthesis.
170	Illegal macro name.
171	Empty formal list in macro definition.
172	More than 31 formals in macro formal list.
173*	Illegal formal parameter.
174*	Macro definition during macro expansion suppressed.
175*	Recursive macro call.

176*	Macro in use at block purge time.
177*	"(" missing on macro call.
200*	Missing exponent on floating constant: 0 assumed.
201	Compile time (floating) division by zero.
400	A special unary operator appeared in the text, and it was not followed by an open parenthesis. The special unary operators are the character manipulation functions (SCANN, REPLACEI, etc.). MACHOP not in declarations, and the special functions SIGN, ABS, FIRSTONE and OFFSET. Since the special unary operators are analogous to function calls, they require an argument list. Since they are special functions, there is no machine address associated with the name; therefore, the name alone has no meaning and is illegal.
401	While evaluating a compound expression or block, we find an expression whose close bracket is "ELSE". This is illegal. BLISS-10 makes the assumption that it was the intent of the writer to close the compound expression before the ELSE was encountered. Therefore, BLISS-10 closes the compound expression on recovery and continue processing at the ELSE operator.
402*	An OTHERWISE or ALWAYS was encountered in a SELECT expression after an NSET element which was not followed by a semicolon as required. BLISS-10 assumes the semicolon and continues as if it were there all along.
403#	The compiler has attempted to generate a machine address for an identifier of an illegal type (such as a special operator). If this problem is encountered, it will most likely be the result of the compiler's failure to detect an earlier syntax error.
404*	A GLOBAL BIND must be to a compile time constant expression. Since this expression is not, we assume that this is a non-global BIND and process it as such.
405	Over 15 formal parameters appear in this structure declaration. 15 is the maximum.
406*#	The compiler can't shrink back to original size after a compilation.
407	This input line contained more than 135 characters. It won't fit into the line buffer. Shorten it and try again.
410*	There is no name preceding this NAMES or INDEXES bind. This NAMES or INDEXES bind is ignored.
411*	This name has already been declared at this block level. This NAMES or INDEXES bind is ignored.

412*	Cannot do NAMES or INDEXES bind in a portion of a PLIT subject to duplication. Attempt at binding is ignored.
413*	This identifier has been declared as a GLOBAL symbol in another context. This GLOBAL declaration is ignored. Non-global declaration still takes place.
414*	The symbol NAMES or INDEXES must follow the symbol GLOBALLY in a PLIT.
415*	On a structure access, the number of actual parameters passed was less than the number of formals in the structure declarations. Zeroes are assumed for missing actuals.
416*	It is illegal to use a label in this context. The label is ignored.
417	The atom following LEAVE is not a proper label.
420*	This label has already been used in this block. This attempt to use it is ignored.
421	We are outside the scope of the expression so labeled; thus this reference is illegal.
422*	The parameter to the OFFSET function must be a local or formal variable name. If not, the value 0 will be returned for the value of the OFFSET function.
423*	This construct is scheduled to be removed from this compiler and possibly the BLISS-10 language at some future date. We suggest that you use an alternate implementation which does not require this construct.
424*	An attempt has been made to declare a name which is identical to a previously declared global type name in the first six characters. The compiler recovers by considering this instance an OWN ROUTINE declaration.
425*	An attempt has been made to declare a name which is identical to a previously declared global in the first six characters. The compiler recovers by considering this instance an OWN declaration.
500	Input error while reading a source file (the name of the source file is printed).
501	Invalid CMU userid, i.e. the CMUDEC algorithm failed.
502	File not found.
503	Invalid PPN format.
504	Device not found or could not do ASCII input.
505	REQUIRE declarations nested more than six levels.

601	Improper switch use.
602	Switches declaration is missing its terminating semicolon.
603	This switch name is unknown by this compiler.
606*	The RESERVE statement in the module head must be bracketed by parentheses, and it is not.
610*	The DREGS switch in the module head must be followed by an equals sign, and it is not.
612	An attempt has been made to use a register name which is not a compile time constant in the range 0-15.
613	Module declaration within module body.
614	Invalid type in stack or timer declaration.
615	Syntax error in stack or timer declaration.
616	Invalid size expression in stack or timer declaration.
617	Trying to reserve a register in use already.
620	Special register register-number not valid.
621	Trying to declare a special register already in use.
622	Module errors; compilation starts at pointer.
623	Number of reserved registers plus number of special registers (4) plus number of declarable registers (DREGS) exceeds 12. This register reservation is, therefore, ignored. The compiler requires at least four temporaries in addition to reserved plus special plus declarable registers.
624	Missing equals in a special register declaration.
625	Module declaration errors with a trivial program.
626	Syntax errors in entries switch.
627	Declared entry point is not defined.
630	Plit missing right parenthesis.
631	Compile time expression error.
632	Load time expression error.
633	Negative PLIT duplication factor.
634	May not use long string in this context.
635	String missing right quote.

636	Invalid escape character in string.
637	Extra actuals passed to structure access.
760	No temporary register available.
761	No declared registers available (INCR/DECR).
762	No declared registers available (declaration).
770	The compiler is unable to acquire the additional core needed for this compilation.
771#	GT SAVEF overflow.
772#	Reg. table use field overflow.
773#	Graph table UCCF overflow.
774#	Literal table capacity exceeded.
775#	Pointer table capacity exceeded.
776	Operand pair without intervening delimiter.
777#	Compiler error.

INDEX

BLISS-10
Programmer's Reference Manual
DEC-10-LBRMA-A-D

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form (see the HOW TO OBTAIN SOFTWARE INFORMATION page).

Did you find errors in this manual? If so, specify by page.

____
____
____
____
____
____

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

____
____
____
____
____
____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

____
____
____
____
____
____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name ______________________ Date ______________

Organization ______________________________

Street ______________________________

City ______________ State __________ Zip Code __________
or
Country

If you do not require a written reply, please check here. ☐

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Communications
P. O. Box F
Maynard, Massachusetts 01754