BLISS
Primer

digital

BLISS
Primer

Intermediate

This course reflects:

COMPILERS: BLISS-16c
                              BLISS-32
                              BLISS-36

Educational Services

Digital Equipment Corporation

Marlboro, Massachusetts

BLISS Primer

LANGUAGE ABSTRACT

MODULE II-1

Course Map

```
                        ╭─────────────╮
                        │  UTILITIES  │
                        ╰─────────────╯


  ╭─────────────╮      ╭──────────────╮      ╭──────────────╮
  │    TABLE    │      │ OPTIMIZATION │      │  DEBUGGING   │
  │   BUILDING  │      ╰──────────────╯      │  TECHNIQUES  │
  ╰─────────────╯             ▲             ╰──────────────╯
        ▲                     │                     ▲
        │                     │                     │
  ╭─────────────╮             │             ╭──────────────╮
  │ ADDITIONAL  │             │             │ CONDITIONAL  │
  │    DATA     │             │             │ COMPILATION  │
  │  STRUCTURES │             │             ╰──────────────╯
  ╰─────────────╯             │                     ▲
        ▲                     │                     │
        │                     │                     │
  ╭─────────────╮      ╭──────────────╮             │
  │    DATA     │◄─────│  CHARACTER   │─────────────╯
  │  STRUCTURES │      │  HANDLING    │
  ╰─────────────╯      │  FUNCTIONS   │
                       ╰──────────────╯
                              ▲
                              │
                       ╭──────────────╮
                       │   LEXICAL    │
                       │  FUNCTIONS   │
                       ╰──────────────╯
                              ▲
                              │
                       ╭──────────────╮
                       │   LANGUAGE   │
                       │   ABSTRACT   │
                       ╰──────────────╯
```

II-1-2
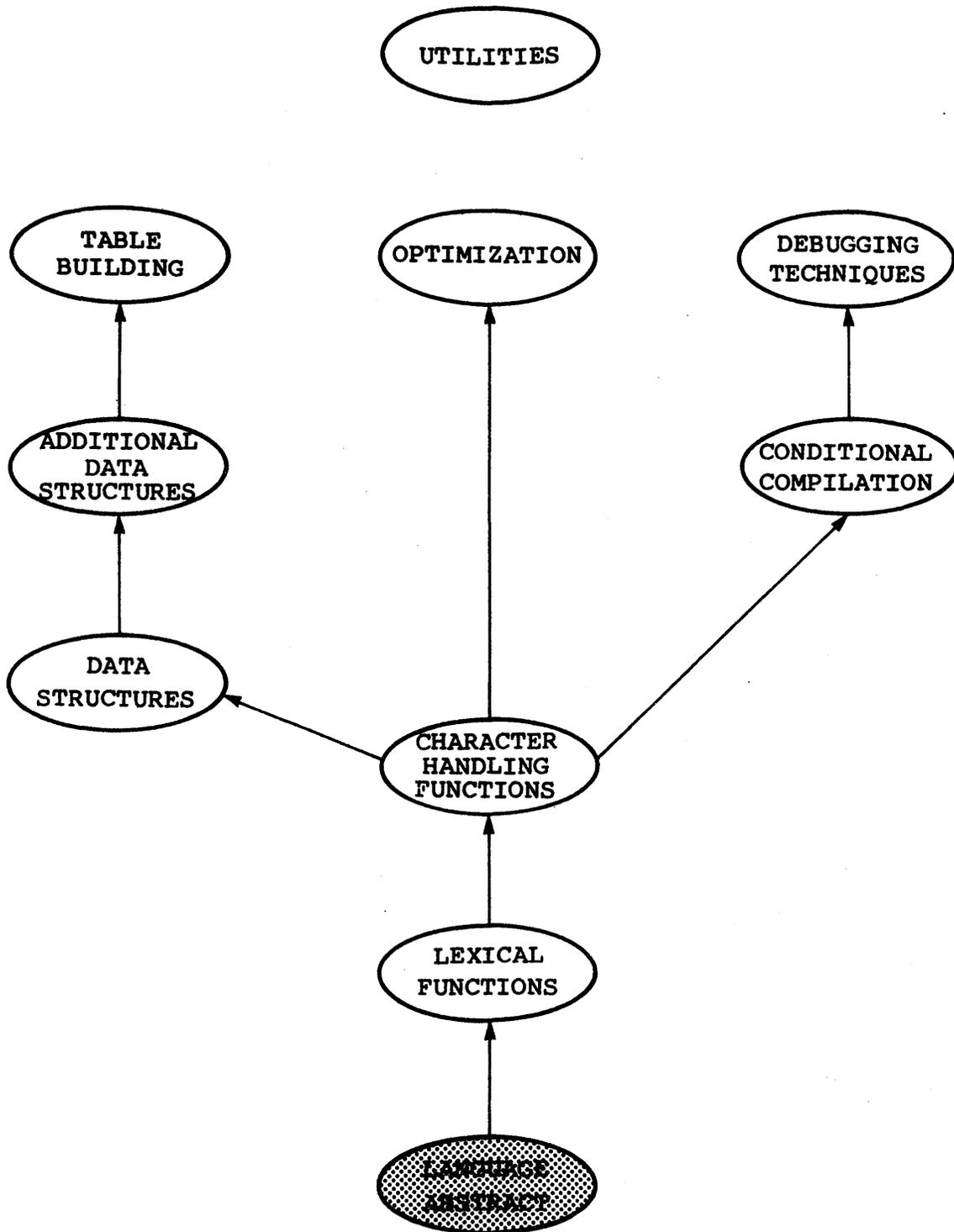
Introduction

This unit presents BLISS from an abstract, conceptual viewpoint. The language is described in terms of a few very general rules and these rules are contrasted with non-BLISS languages. The presentation assumes a familiarity with BLISS syntax as presented in the prerequisite units.

```
┌─────────────────────────Objectives──────────────────────────────┐
│                                                                  │
│     Given an arbitrary BLISS expression using contructs from  the│
│ prerequisite  units,  be  able to determine if it is valid and, if│
│ so, its value.                                                   │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

```
┌──────────────────────Sample Test Items──────────────────────────┐
│                                                                  │
│     Given that  the  location  A  is  initially  zero,  determine│
│ whether  the  following  expressions  are  valid and, if so, their│
│ value.                                                           │
│                                                                  │
│     1)   X = INCR COUNT FROM 0 TO 4  DO                          │
│              A = .A + 1;                                         │
│                                                                  │
│     2)   (IF .A GTR 0 THEN 2 ELSE 5 ) = .A;                      │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

```
┌─────────────────────Additional Resources────────────────────────┐
│                                                                  │
│ None.                                                            │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

GENERAL

This primer has thus far de-emphasized the interaction of expressions in order to emphasize the fundamental concepts of the language. Now that a substantial subset of the syntax has been presented, it is appropriate to examine the general semantics of the language. From a conceptual viewpoint, the language can be described with 2 general rules:

1)   Every expression has a value.
2)   The context of a value determines its meaning.

However, the characteristics of the language which allow this simple generalization and the characteristics which make the language versatile also create the most confusion in the minds of new users. This is why a specific discussion of these rules has been deferred until now.

BLISS IS AN EXPRESSION LANGUAGE

The statement "BLISS is an expression language" can usually be better appreciated by programmers familiar with an expression language such as LISP, than by programmers familiar with non-expression languages such as FORTRAN. This is at least partially true because users who are not familiar with an expression language tend to associate the term "statement" with the term "expression". As will become apparent, statements and expressions are not at all synonymous.

BLISS has three types of expressions: primary, control, and compound. Since primary expressions have not previously been explicitly identified as such, they are depicted below with an example of each:

Primary Expressions	Examples
literal	3
plit	PLIT(%ASCIZ'STR',5)
name	SYM_TABLE
routine call	EXCHANGE(X,Y)
block (where the "e"s are expressions)	(e1; e2; e3)
structure reference	TABLE[5]

Of course, most of these primaries are also available in
other languages; however, in non-expression languages they
frequently have meaning only in specific contexts. In BLISS they
can be used anywhere an expression is permitted. This distinction
is very apparent when comparing the allowable expressions in BLISS
and non-BLISS control constructs.

Control constructs include conditional, case, select, and
loop expressions. Although most languages have one or more
equivalent constructs, a comparison between expression and
non-expression languages as to the type of expression permitted in
the various control constructs emphasizes a fundamental
difference:

BLISS	Non-Expression
IF expression THEN expression ELSE expression	IF Boolean-construct THEN statement ELSE statement

WHILE expression DO
    expression
	WHILE Boolean-construct DO
	    statement

INCR name
    FROM expression
    TO expression
    BY expression
    DO expression
	INCR name
	    FROM arithmetic-statement
	    TO arithmetic-statement
	    BY arithmetic-statement
	    DO statement

In BLISS, an expression is an expression is an expression. That
is to say, there is, in general, no restriction on the type of
expression which can be used in the various parts of a control
expression. You can, for example, use another control expression
as the test condition, permitting constructions like:

```
IF (IF .A THEN ... ELSE ...)
THEN ...
ELSE ...
```

This is not true in many other languages. For example, in other
languages, it is common to allow only Boolean constructs in
conditional tests (after IF or WHILE), and arithmetic constructs
where numeric values are anticipated (after FROM, TO, BY, etc.).
The more general BLISS syntax has many advantages. For example,

```
    IF (IF .P NEQ Ø
        THEN ..P GTR Ø
        ELSE Ø)
    THEN ...
    ELSE ...
```

forces the test of the pointer P to be performed before using it;
whereas

```
    IF .P NEQ Ø AND
        ..P GTR Ø
    THEN ...
    ELSE ...
```

may not perform the test ".P NEQ Ø" first and therefore could
result in an illegal memory reference on some systems. This
conceptual difference between BLISS and non-BLISS-like languages
also extends to the creation of compound expressions.

Compound expressions result from combining primary and/or
control expressions with assignment, arithmetic, Boolean, or
relational operators. Compound constructs in BLISS and non-BLISS
languages have the general forms shown below:

BLISS
_____

expression OPERATOR expression
(control-expression) OPERATOR (control-expression)

non-BLISS
_____

arithmetic-expression OPERATOR arithmetic-expression
name = arithmetic-expression

Note that there are two forms shown for the BLISS and the
non-BLISS languages.   The second form in each is an exception to
the more general rule above it: i.e., in BLISS, control
expressions used where a value is required must be enclosed in a
block (in this case parentheses);   and, in most non-BLISS
languages, explicit names are usually required on the left side of
an assignment operator. The BLISS exception (control expressions)
is illustrated by the following examples:

```
    X = (IF condition
        THEN...
        ELSE ...);
```

and

```
X = (SELECTONE index OF
     SET
     [label 1]: expression 1;
     ...

     [label n]: expression n;
     TES);
```

Note: When writing production programs, the VAX-11 Software
Engineering Manual should be consulted for acceptable coding
practices. This is necessary because the misuse of the language,
through valid but exotic expressions, creates more problems than
it solves: although BLISS will compile exotic constructs, they
may be confusing to readers of the code and defeat the purpose of
BLISS as an implementation language, or they maybe be confusing to
the user and require substantial debugging time.

These examples should suggest both the relative simplicity of
BLISS, taken in the abstract, and the complexity of the
expressions which the language permits you to construct.

In order to achieve this generality which permits, among
other things, the arbitrary nesting of control constructs, each
expression must return a value when used in a context which
requires one.

EVERY EXPRESSION HAS A VALUE

This is perhaps the most significant distinction between an
expression language like BLISS and non-expression languages.
Non-expression languages tend to be comprised of "statements"
which, in general, describe actions to be taken but do not have a
value as such. These statements merely perform an operation or
step. For example, the assignment

```
TEMP = 256;
```

would store the value 256 at the symbolic address TEMP; however,
in non-expression languages the statement itself would have no
value. As a result,

```
INCR COUNT FROM 0 TO TEMP = 256 DO
    BUFF[.COUNT] = TTY_GET_CHAR();
```

would be meaningless. Non-expression languages therefore permit

only arithmetic expressions in such contexts, since they are the
only statements that produce a value. In BLISS an assignment
expression (as a entity) has a value, and that value is the same
as the value being stored. This means in BLISS that the above
example would be equivalent to:

```
INCR COUNT FROM 0 TO 256 DO
     BUFF[.COUNT] = TTY_GET_CHAR();
```

with the additional "side-effect" of storing the value 256 in
location TEMP. It is therefore common to combine the first test
of a value with its assignment to a temporary, e.g.,

```
IF (TEMP = TTY_GET_CHAR()) GEQ %O'40'
THEN ...
```

The value of an assignment expression is in fact a fullword
bit pattern and need not be a numeric value (logically) as such.
For example, the assignment

```
TEMP = PLIT(%ASCIZ'A MESSAGE');
```

has as its value the address of the PLIT. Since the assignment
operator associates to the right, this value could in fact be
"reused" by adding yet another level of assignment,

```
PTR = TEMP = PLIT(%ASCIZ'A MESSAGE');
```

and so on.

Rules for determining the value of common primary and
operator expressions include:

* The value of a PLIT is its address.
* The value of a name is its address.
* The value of the fetch operator (".") applied to a name
  is the contents of the corresponding memory location.
* The value of a routine call is the value returned
  by the called routine.
* The value of a block is the value of the last expression
  executed in the block.
* The value of a compound expression made from a
  relational operator is 1, if true, and 0, if false.
* The result of a compound expression made from a
  Boolean operator is the resulting bit pattern.

These are relatively straightforward results with the possible
exception of a NOVALUE routine call. Because no value is

returned, NOVALUE routine calls are therefore not permitted (they are invalid) in any context which requires a value. For example, in the following expressions the NOVALUE routine EXCHANGE is used in an erroneous and invalid manner:

```
EXTERNAL ROUTINE
     EXCHANGE: NOVALUE;
...

A = EXCHANGE(X,Y) + 20;                    ! INVALID EXPRESSION
...

IF EXCHANGE(X,Y)                           ! INVALID EXPRESSION
THEN expression
ELSE expression;
```

It is important to note however that these expressions would be correct if the routine EXCHANGE returned a value.

The rules governing the value of control expressions are somewhat less intuitive. They include:

* For SELECT and SELECTONE:
    the value of the last (or only) expression evaluated
    or -1 if none are selected.
* For CASE:
    the value of the expression executed
    or undefined if no match is found.
* For loops:
    -1 on normal completion or the value of the LEAVE
    or EXITLOOP. expression that terminated the loop.
* For IF-THEN-ELSE:
    the value of the THEN or ELSE expression
    SELECTed.

Note that although the defined value of -1 for SELECT/SELECTONE and loop expressions may seem arbitrary, it does provide a means of determining the action taken during execution. For example:

```
TYPE = BEGIN
        SELECTONE (CHAR = TTY_GET_CHAR()) OF
          SET
          [%C'A' TO %C'Z']:   ALPHA;
          [%C'0' TO %C'9']:   NUMERIC;
          TES;
        END;
```

which will read a character, assign it to CHAR, and return in TYPE:

the value of ALPHA if an alphabetic character
the value of NUMERIC if a numberic character
the value -1 if not alphanumeric

The value of any expression is a fullword bit pattern which can be determined by applying the above rules; however, the meaning associated to that value depends entirely upon the context in which the value is used, and not on the way it was generated.

THE MEANING OF A VALUE

The meaning of a value is determined by the context in which it is used, not by any intrinsic quality of the expression which produces the value. Consequently, the expressions

```
TEMP = %C'S';
TEMP = SAM;
TEMP = 10 * 8 + 3;
```

would all have exactly the same value, assuming the address of SAM was decimal 83. In many non-expression languages, special declarations (implicit or explicit) are necessary to store numbers, strings, and addresses since they are thought of as being different kinds of data. Furthermore, identical expressions can have different values depending upon the context in which they are used. For example, the assignment:

```
A = A + 1;
```

In most non-BLISS-like languages the value of A on the left of the assignment operator is its address, whereas the value of A on the right of the assignment operator is its contents. In BLISS the value of a symbolic name is always the same, i.e., an address. Hence, in this example A would be assigned the address of the next memory location - not one more than its present contents. To add one to the contents of A would require the use of the "dot" operator, as follows:

```
A = .A + 1;
```

Although this unit can not discuss all of the implications resulting from these three BLISS rules (abstracted above), a few of the more significant considerations are presented below.

FURTHER RAMIFICATIONS

A common programming requirement is to assign one of two values to a given location depending upon the results of some test. This operation could be diagrammed as,
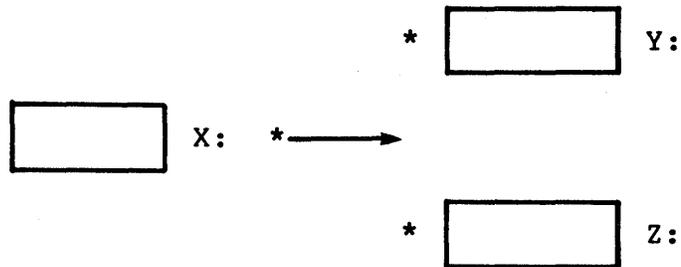
where the condition (depicted by the arrow) acts as a switch to direct either the contents of Y or Z to be transferred to X. The code in most languages would be equivalent to:

```
IF condition
THEN
    X = .Y
ELSE
    X = .Z;
```

In BLISS this same sequence can be written equivalently as:

```
X = (IF condition THEN .Y ELSE .Z);
```

The value of the IF statement becomes the value of the selected THEN or ELSE part (i.e., the contents of Y or Z), and this value is in turn stored in X. The syntax corresponds very closely to the diagrammatic logic of the situation. It is important to again note that the control expression has been enclosed in parentheses. This restriction (i.e., having to enclose control expressions - used as a value - in a block) is imposed for the user's benefit. It is necessary because of the ambiguity which can and frequently does result without an explicit indication of the end of the ELSE expression. For example, the expression

```
X = IF A GTR B THEN A ELSE B+4;
```

has two possible interpretations:

```
X = (IF A GTR B THEN A ELSE B) + 4;
X = (IF A GTR B THEN A ELSE B + 4);
```

Another common programming requirement is to assign a value to one of two locations, depending on the result of some test. The resulting construct in most languages could be illustrated as:

```
┌──────────┐   Y:  *
└──────────┘


              ◄───────*      ┌──────────┐   X:
                             └──────────┘

┌──────────┐   Z:  *
└──────────┘
```
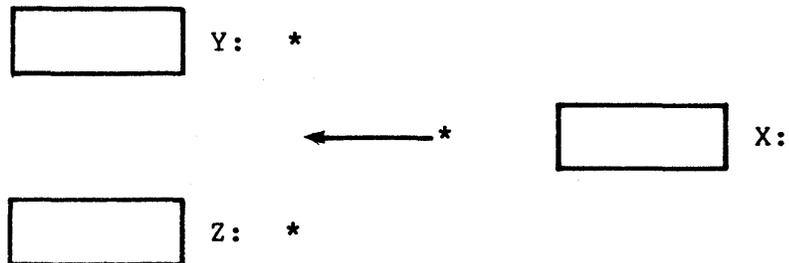
This would be programmed in most languages as:

```
IF condition
THEN
     Y = .X
ELSE
     Z = .X;
```

BLISS, on the other hand, permits the more direct construction:

```
(IF condition THEN Y ELSE Z) = .X;
```

Observe that the parentheses in this example are essential to achieve the intended result, as the expression would be interpreted differently without them.

Expressions in BLISS may be arbitrarily complex and nested to great depths. The block is a good example:

```
BEGIN                                   \
declarations;...
expressions;...
END
```

or equivalently,

```
(declarations;... expressions;...)
```

The block is itself a primary expression, but it can be composed of many more blocks — each with many sub-expressions. For instance,

```
(el; e2; e3;... (ell; el2; el3; eln);... en)
```

where each block may also contain its own separate declarations.

It is in fact common practice in BLISS to declare local variables
or macros for an inner block. For example:

```
IF ...
THEN
     BEGIN
     LOCAL.
         X,
        , Y;
     ...
     X = ...
     Y = ...
     END
ELSE ...
```

Exercises

Given:

```
A = 0
B = %C'S'
POS() - a routine which always returns the value 1
EXCH() - a NOVALUE routine
```

For each expression below, determine its value if valid, or the reason it is invalid.

1)   A = 25 - 5;

2)   A = (B = 5; C = .B + 5; D = 0);

3)   A = (IF POS() THEN A = 6 ELSE A = 4);

4)   A = (IF .B THEN 7);

5)   INCR COUNT FROM 0 TO 4 DO
            A = .A + 1;

6)   A = %C'B';

7)   A = EXCH();

8)   A = IF POS() THEN .B ELSE .A;

9)   A = (SELECT .B OF
            SET
            [%C'S']   : 1;
            [%C'T']   : 2;
            [%C'O']   : 3;
            [ALWAYS]   : 4;
            TES);

10)   .(A + 5) = 16;

11)   7 = (INCR COUNT FROM 0 TO 100 DO
                IF .COUNT GTR 50
                THEN
                    EXITLOOP 5
                ELSE
                    A = .A + 1);

12) A = .POS();

13) A = (.B + 1;);

This page intentionally left blank.

Solutions

1)   20

2)   0

3)   6

4)   invalid   (the ELSE part is required when an IF control
                     expression is used as a value)

5)   -1   (normal termination)

6)   %C'B'   (or octal 102)

7)   invalid   (this context requires a value)

8)   invalid   (control expressions require parentheses when
                     used in a context which requires a value -
                     such as in this assignment)

9)   4   (ALWAYS is last label selected)

10)   16

11)   5   (the value of EXITLOOP - it is always a
                     premature exit)

12)   contents of location one

13)   0   (the expression is equivalent to: A=(.B+1;0);)

This page intentionally left blank.

Unit Test

1. Given that the contents of A is initially zero in each expression below, determine whether each expression is valid and, if so, its value:

a)   X = (WHILE .A GTR 15 DO
            A = .A + 1);

b)   (IF .A GTR 0 THEN 2 ELSE 5) = .A;

c)   X = INCR COUNT FROM 0 TO 4 DO
            A = .A + 1;

2. Given:

    the contents of B = %C'X'
    the address of B = 112

For each question below, determine the value of each expression:

a)   SELECT .B OF
            SET
            [%C'B']   : A = 1;
   [%C'O']   : A = 2;
            [%C'X']   : B = 3;
            [OTHERWISE]   : B = 4;
            TES;

b)   TEMP = B;

c)   B AND .B;

d)   .B GTR B;

This page intentionally left blank.

LEXICAL FUNCTIONS

MODULE II-2

Course Map

```
                        ⬭ UTILITIES ⬭


    ⬭ TABLE        ⬭ OPTIMIZATION ⬭        ⬭ DEBUGGING
      BUILDING ⬭                             TECHNIQUES ⬭
        ↑                ↑                        ↑
    ⬭ ADDITIONAL                            ⬭ CONDITIONAL
      DATA                                    COMPILATION ⬭
      STRUCTURES ⬭
        ↑                                       ↗
    ⬭ DATA                                 
      STRUCTURES ⬭ ←   ⬭ CHARACTER
                          HANDLING
                          FUNCTIONS ⬭
                             ↑
                       ⬭ LEXICAL
                         FUNCTIONS ⬭
                             ↑
                       ⬭ LANGUAGE
                         ABSTRACT ⬭
```
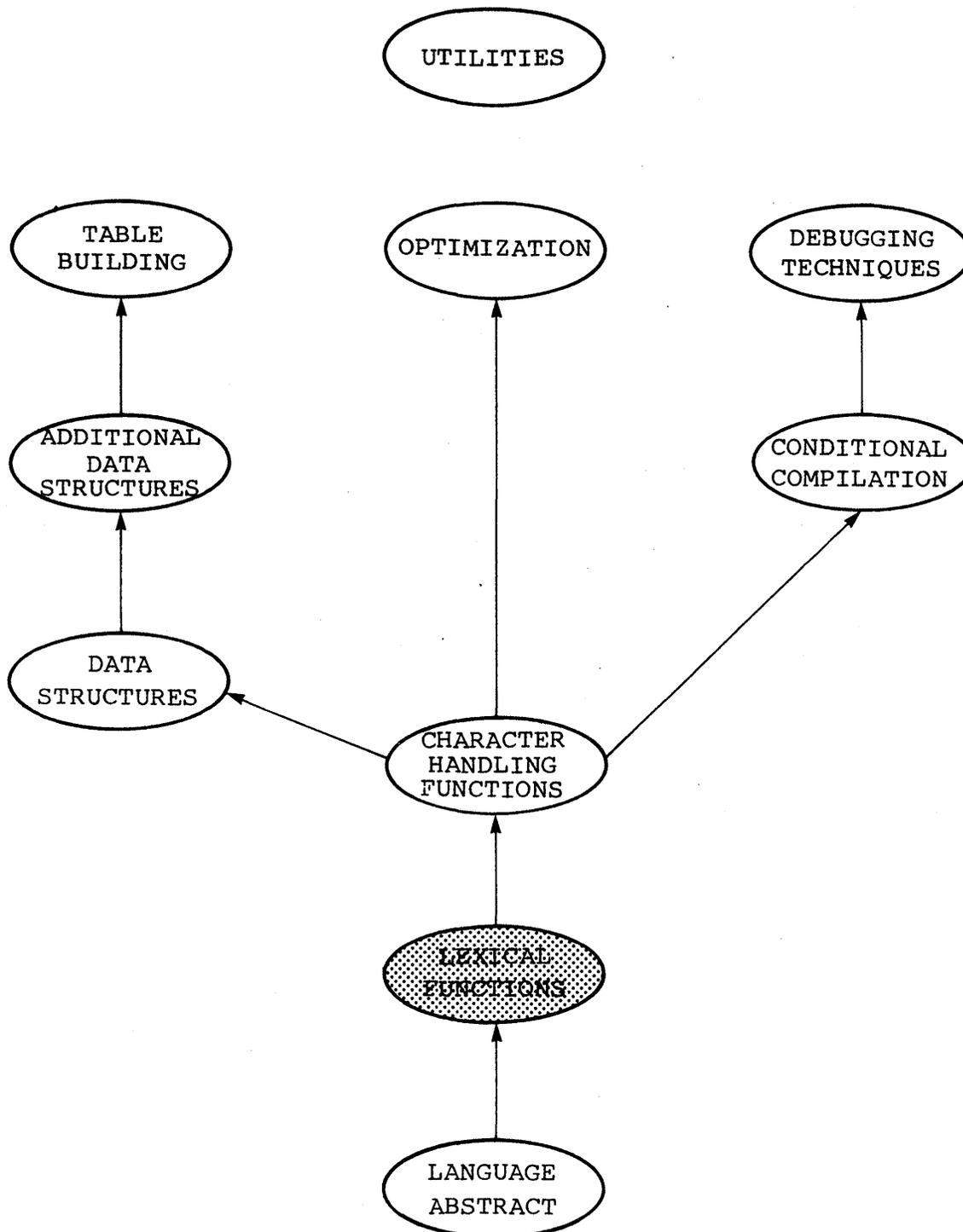
Introduction

The lexical string, protect, and macro functions are introduced along with examples. These functions are primarily used with macros although the string functions can be used to simplify certain string processing operations. Conditional macros are also discussed.

Objectives

Be able to use arbitrary lexical functions correctly in the solution to a specified problem involving conditional macros.

Sample Test Items

Write a program, using a conditional macro named AVG, that accepts an arbitrary number of integer arguments and subsequently computes their average. The solution must use at least four different lexical functions.

Additional Resources

BLISS-32 Language Guide   Chapter 16: Lexical Processing
                                 Section: 16.2

Lexical functions are a means of producing or controlling substitutions in the input stream to the compiler. As such, these functions allow the user to create or manipulate literals and symbols in ways which would not otherwise be permitted.

One case of lexical substitution occurs in a macro expansion when the macro name (and its arguments) is removed from the input, and the text of the macro is substituted in its place. The final result is as if the text had been typed in the original source code.

Although the lexical string functions have some direct application outside of macros, lexical functions are primarily used in conjunction with the macro facility. Since several of the lexical functions have special applications in conditional macros, the latter will be discussed before proceeding to specific lexical functions.

CONDITIONAL MACROS

A conditional macro has the general form

```
MACRO
     name(formal-parameters,...)[]=
         text %;
```

with the expansion contingent upon both the number of formal parameters in the macro definition and the number of actual arguments in the macro call. In general, expansion occurs as long as the number of actual arguments in the macro call equals or exceeds the number of formal parameters in the macro definition. This means that if you define a conditional macro as having two parameters and then call it with five actual parameters, the macro will be expanded twice: once for the first two arguments and once for the second set. No expansion occurs for the third set because only one of the two required arguments was supplied. Examples will be deferred until the specific lexical functions which normally occur in conditional macros are discussed.

STRING FUNCTIONS

The lexical string functions include:

%CHAR
%CHARCOUNT

%NAME
%NUMBER
%STRING

The function %CHAR(integer,...) accepts a series of one or more integers, separated by commas, and produces a concatenated string of ASCII characters. For example (using 32-bit architecture):

A = %CHAR(79,88);

```
+---------------------------+
|                   X  O    | A:
+---------------------------+
```


The function %CHARCOUNT(string,...) accepts a series of string literals, separated by commas, and produces a count of the number of characters in the concatenated strings. For example:

A = %CHARCOUNT('A','LONG','STRING');

```
+---------------------------+
|                       11  | A:
+---------------------------+
```


The function %NAME(string,...) accepts a series of string literals and returns a concatenated string that is interpreted as a name. This function is necessary to reference names containing special characters. For example,

```
MACRO SYSCALL =
     %NAME('.$SYSCALL') %;
```

constructs an equivalent name for an external systems service routine name that begins with a period, even though the period has other meanings in BLISS. Using the macro name, the service routine can then be referenced in a normal manner as shown below:

```
EXTERNAL ROUTINE
     SYSCALL;
...

SYSCALL();
```

Since this function accepts multiple string arguments, it is also used to construct new names. For example,

```
MACRO LOCAL_SYM(A)=
     LOCAL
          %NAME('LOC_',(A)) %;
...

LOCAL_SYM('VEC');
```

concatenates the strings LOC and VEC to produce the declaration:

```
LOCAL
    LOC_VEC;
```

The function %NUMBER(item) accepts either a string literal or a literal name and returns the appropriate decimal value. For example:

```
LITERAL
    TYPE_A = 3;
...

A = %NUMBER(TYPE_A);

A = %NUMBER('65');
```

3
65

The function %NUMBER has meaningful application only in conjunction with macros. For example,

```
MACRO CTRST(A,B) =
    OWN %NAME((A), %NUMBER(B)) %;
```

accepts a literal name or a numeric literal and makes possible macro calls of the form:

```
LITERAL
    TYPE_A = 3;
...

CTRST('LVAL_',TYPE_A);
```

to produce:

```
OWN
    LVAL_3;
```

The last lexical string function is %STRING(string,...) which concatenates multiple string literals, separated by commas, into a single string literal. This function is useful in creating strings which require special characters that could not otherwise be incorporated. For instance,

```
TTY_PUT_QUO(%STRING('EXIT LINE ',%CHAR(%O'15',%O'12')));
```

would output to the terminal the string "EXIT LINE " followed by a carriage return and line feed. As shown in the above example, lexical functions may be nested arbitrarily, one inside the other.

It is important to emphasize that lexical functions effect a substitution by the processor which occurs before compilation. This is depicted below using a few of the previous examples:

Before Lexical Processing   After Lexical Processing
_____   _____

    A = %CHAR(79,88);   A = 'OX';
    A = %CHARCOUNT('A','LONG','STRING');   A = 11;

Therefore an expression like

    A = %CHAR(.TEMP);

is invalid because ".TEMP" (the contents of location TEMP) has no value until run-time.

In general, there are three types of lexemes in a macro:

    string literals   (i.e., %C'A', 'GO', etc)
    names   (i.e., A, FOO, etc.)
    numeric literals   (i.e., 1, 345, etc.)

In conjunction with macros, it is often necessary or desirable to convert from one type of lexeme to another when the type of the desired result is known but the type of the argument is not. The following table summarizes the possible arguments and their result:

Function	Argument type	Result
%STRING	(string-literal)	same
	(name)	concatenation of each character in the name taken in textual order
	(numeric-literal)	the character representation of the numeric literal
%NAME	(string-literal)	name corresponding to the string
	(name)	same
	(numeric-literal)	%NAME(%STRING(numeric-literal))

%NUMBER   (string-literal)   %DECIMAL'string-literal'

                         (literal name)   value of the literal
                                                      name

PROTECT FUNCTIONS

Without the protect functions, many useful applications of
the macro facility would be made more difficult if not impossible.
For example, an attempt to pass, in a macro call, an argument
which contained a comma, as in

```
FUNC(A,B);
FUNC(,);
```

would not succeed. The first call would be interpreted as two
arguments (A and B) rather than one (A,B), and the second would
also have two parameters, both null. Furthermore, it would not be
possible to nest macro declarations since the attempt to end the
inner macro declaration with a "%" would terminate the outermost
macro and generate subsequent compiler errors because the leftover
code would not be syntactically correct. The following nested
macros illustrate this problem:

```
MACRO FMA(X) =
...

    MACRO FMB = text %;
    ...

            %;
```

The lexical functions %QUOTE and %UNQUOTE are therefore expedient.
The function %QUOTE prevents the next lexeme (i.e., an entity such
as a keyword, name, literal, string, delimiter, etc.) from being
interpreted (bound) until it is expanded at the time of the macro
call. This function provides the needed mechanism to pass
parameters that contain a comma or another delimiting character as
part of the argument. For example, the macro calls on FUNC above
can now be correctly written as:

```
FUNC(A %QUOTE, B);
FUNC(%QUOTE,);
```

Similarily, the nested macros above would be written:

```
MACRO FMA(X) =
...

    MACRO FMB = text %QUOTE %;
```

```
...
```

```
%;
```

The function %UNQUOTE has the opposite effect. That is, the
lexeme following the function is lexically bound in the current
block. This is used to associate a name to the declaration in
effect within the current block. Normally, a name is not bound to
an address until the time that the macro is expanded; at that
time, the name is associated with the appropriate declaration in
effect in that block. The following code demonstrates this
application:

```
BEGIN
OWN
    X;

MACRO MES (A) =
    BEGIN
    LOCAL
        X;
    X = PROC (.A);
    END; %;

    BEGIN
    OWN
        X;
    MES (X);
    END;

MES (X);
END;
```

This example has two calls on the macro MES, both in different
blocks. Since the name X has been redeclared twice, three
distinct locations for X are possible; however, X will always
refer to the local declaration of X in the macro. To force X to
refer to the declaration in effect within the block declared, the
function %UNQUOTE can be used. Using %UNQUOTE in the call causes
X to be bound to the declaration in effect in the block the macro
was declared. For example,

```
BEGIN
OWN
    X;

MACRO MES (A) =
    LOCAL
```

```
        X;
     X = PROC(.A) %;

   MES(%UNQUOTE X);
   END;
```

causes the argument A in the macro to refer to the declaration of X contained in the call, rather than the declaration of X in the macro.

MACRO FUNCTIONS

The macro lexical functions are defined only for use in the macro body itself. They include these three functions:

```
   %COUNT
   %LENGTH
   %REMAINING
```

The function %REMAINING is frequently used with conditional macro calls because it provides a means to obtain the actual arguments in excess of the number declared. The value of the function is the list of arguments that are in excess of those required for the present copy (incarnation) of the macro. Consider, for example, the recursive macro below which has two formal parameters (X and Y):

```
   MACRO MAC(X,Y)[] =
       X = ALPHA(Y);
       MAC(%REMAINING)%;
```

If this macro were called with four actual arguments as in,

```
   MAC(A,B,C,D);
```

it would produce as its first copy

```
   A = ALPHA(B);
   MAC(%REMAINING)
```

(where %REMAINING equals "C,D") then as its second copy

```
   A = ALPHA(B);
   C = ALPHA(D);
   MAC(%REMAINING)
```

where %REMAINING equals null. Since the final call to MAC has

fewer arguments than the formal parameters declared, the value of the macro call is null, and the final result of the call would be:

```
A = ALPHA(B);
C = ALPHA(D);
```

The %COUNT function is also used with both recursive and conditional macros and returns the number of the current copy of the macro. The orginal is given number, 0, the first copy, 1, etc. For example, the conditional macro

```
MACRO IUC(X)[] =
    FUNC_IUC(X,%COUNT);
    IUC(%REMAINING)%;
```

generates a new copy for each argument in the macro call, and for each copy, %COUNT is incremented and that value substituted for %COUNT. Hence, the macro call

```
IUC(A,B,C);
```

expands as:

```
FUNC_IUC(A,0);
FUNC_IUC(B,1);
FUNC_IUC(C,2);
```

The function %LENGTH on the other hand returns as its value the number of actual parameters passed in that call. For example,

```
MACRO MMC(X,Y,Z) =
    IF %LENGTH NEQ 3
    THEN
        TTY_PUT_QUO('REQUIRES THREE ARGUMENTS...')
    ELSE
        BEGIN
        AVERAGE = .X + .Y + .Z;
        AVERAGE = .AVERAGE / 3;
        END %;
```

when called with

```
MMC(A,B,C);
```

would produce:

```
AVERAGE = .A + .B + .C;
AVERAGE = .AVERAGE / 3;
```

Note that the IF expression is not generated in the final form of
this example. Since %LENGTH is a lexical function, the macro
expansion at compile time for the above call would be:

```
IF  3 NEQ 3
THEN
    TTY_PUT_QUO('REQUIRES THREE ARGUMENTS...')
ELSE
    BEGIN
    AVERAGE = .A + .B + .C;
    AVERAGE = .AVERAGE / 3;
    END;
```

Because the test condition in this case (3 NEQ 3) is a constant
zero (i.e., false), only the code for the ELSE part would be
generated. In general, you can expect a conditional expression to
disappear without warning if the test condition is constant. This
results from optimizations that occur at compile time.

This page is for notes.

Exercises

1. Given the macro:

```
MACRO CEX(A,B) [] =
    A = .A + %COUNT;
    A = BETA(B) / %LENGTH;
    CEX(%REMAINING)%;
```

Determine the expansion which would result from the macro call:

```
CEX(R, S %QUOTE, 3, T, W)
```

2. Determine the output to the TTY of the following expression:

```
INCR COUNT FROM %CHAR(%O'60') TO %CHAR(%O'66') BY 2 DO
    TTY_PUT_CHAR(.COUNT);
```

3. Determine the value of the expression:

```
%CHARCOUNT(%CHAR(12,13,14,15));
```

This page intentionally left blank.

Solutions

1)   R = .R + Ø;
     R = BETA(S,3) / 4;
     T = .T + 1;
     T = BETA(W) / 2;

2)   Ø246

3)   4

This page intentionally left blank.

Unit Test

Write a program, using a conditional macro named AVG, that accepts an arbitrary number of integer arguments and subsequently computes their average which is stored at the predeclared symbolic address AVERAGE. Obtain the average by dividing the sum of the arguments by the number of arguments (you do not have to round-off the result). Show a call to AVG which directs the output to the terminal. The solution must use at least four different lexical functions.

This page intentionally left blank.

CHARACTER HANDLING FUNCTIONS

MODULE II-3

Course Map

UTILITIES

TABLE
BUILDING

OPTIMIZATION

DEBUGGING
TECHNIQUES

ADDITIONAL
DATA
STRUCTURES

CONDITIONAL
COMPILATION

DATA
STRUCTURES

CHARACTER
HANDLING
FUNCTIONS

LEXICAL
FUNCTIONS

LANGUAGE
ABSTRACT
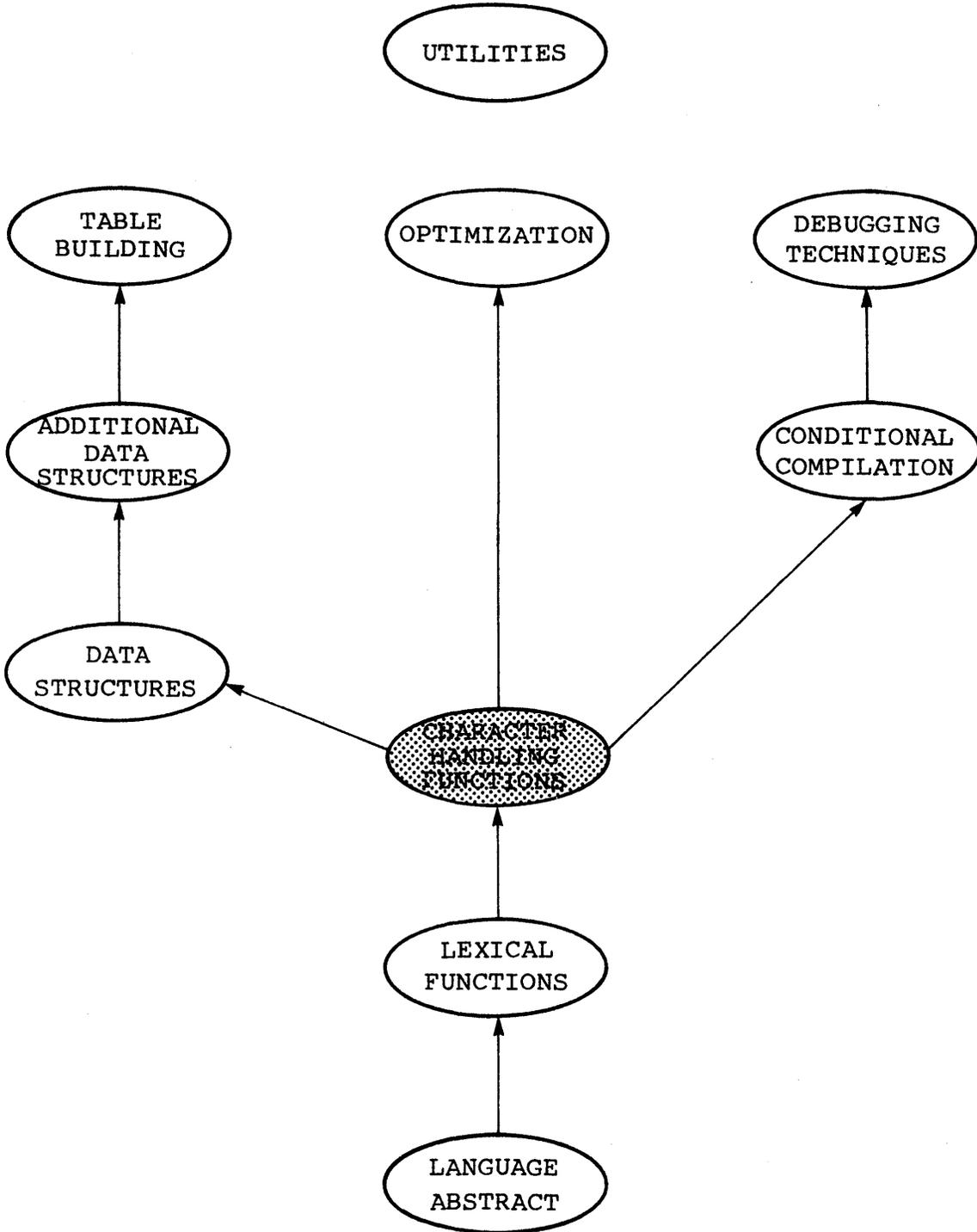
Introduction

The built-in character handling functions are introduced. These functions allow the user to build and manipulate character sequence pointers; to manipulate, compare, search, and move character strings; and to perform character translation.

Objectives

Given a problem which requires the manipulation of character sequences, be able to write a solution which correctly uses an arbitrary number of character handling functions.

Sample Test Items

Write a routine which, when passed the address of an ASCIZ string (having less than 1000 characters) and the address of a work area:

1) translates each character in the string into its Radix-50 equivalent
2) stores the resulting translation in a work area
3) terminates the string with a null character

The solution should use at least 5 different character handling functions.

Additional Resources

BLISS-32 Language Guide   Chapter 20: Character Handling
                                                   Functions
                                       Sections: all

The character handling functions are built-in functions. As such, the syntax for invoking them is the same as that for calling a routine. An actual routine call, however, does not necessarily take place; usually, the necessary code will be generated in-line, saving the overhead of a routine call.

These functions are the preferred method of dealing with characters in BLISS, replacing most of the direct manipulation techniques employed in the examples and exercises throughout the introductory units. They provide code efficiencies as well as being functionally equivalent between machine types.

The character functions are discussed below in groups of the general class of facilities they provide.

FUNCTIONS USED IN ALLOCATING STORAGE

A character sequence, sometimes referred to as a "string", is any series of coded characters stored contiguously in memory. Allocating storage to hold such character sequences is ordinarily the responsibility of the user. To assist the user, two functions have been provided:

CH$ALLOCATION
CH$SIZE

The function CH$ALLOCATION(length, character-size) returns the number of fullwords needed to represent a sequence of a given character length (in number of characters) and a specified size in bits per character. The returned value is:

(number-of-characters + 3) / (characters-per-fullword)

For example, if storage for a character sequence is allocated by means of a vector and the length of the sequence is 45 characters then

```
     OWN
         CH_STORE:VECTOR[CH$ALLOCATION(45)];
```

will produce the correct storage allocation. Character size (the second argument) is optional and has the following defaults:

```
          BLISS-32.....IS A CONSTANT:  8
          BLISS-16.....IS A CONSTANT:  8
          BLISS-36.....VARIABLE FROM 1 TO 36 BITS (DEFAULT: 7)
```

The function CH$SIZE(source-pointer) returns the number of bits per character. FUNCTIONS THAT BUILD AND MANIPULATE CS-POINTERS

A character sequence is specified by two numbers: a character sequence pointer (CS-pointer) which points to the start of the sequence, and its length in number of characters. The three functions which deal with character sequence pointers are:

```
CH$PTR
CH$PLUS
CH$DIFF
```

The function

```
CH$PTR(address, position-index, character-size)
```

builds a CS-pointer. It returns as its value a CS-pointer which represents the beginning address of the sequence plus an offset to the specific character in the sequence. The position index is the offset, or number of character positions from the initial address, and if not specified, it defaults to zero; size is the number of bits per character, and it defaults to eight (BLISS-16/32) or to seven (BLISS-36).

The function

```
CH$PLUS(CS-pointer, index)
```

performs logical addition on the CS-pointer, incrementing it by the specified number of characters. It returns a CS-pointer to the subsequent beginning, now "index" number of characters from the original CS-pointer.

The function

```
CH$DIFF(CS-pointer-1, CS-pointer-2)
```

performs logical subtraction on a pair of CS-pointers and returns an integer index. For example:

```
OWN
    CSP,                          !POINTER STORAGE
    CSP2,                         !POINTER STORAGE
    INDEX;                        !OTHER STORAGE

CSP = CH$PTR(PLIT('EXAMPLE'));    !CSP CONTAINS A
                                  ! CS-POINTER WHICH
                                  ! POINTS TO THE INITIAL
```

```
                                          ! E IN 'EXAMPLE'

    CSP2 = CH$PLUS(.CSP, 5);              !CSP2 CONTAINS A
                                          ! CS-POINTER WHICH
                                          ! POINTS TO THE
                                          ! L IN 'EXAMPLE'

    INDEX = CH$DIFF(.CSP, .CSP2);         !INDEX CONTAINS INTEGER
                                          ! 5
```

Many functions return a CS-pointer. The function CH$PTR creates a CS-pointer while other functions, such as CH$PLUS, return an updated CS-pointer. In either event, it is the responsibility of the user to insure that the correct length stays associated with each CS-pointer. This is essential because the length of the source and/or destination pointer is required as an input argument in several of the following character handling functions.

FUNCTIONS WHICH MANIPULATE CHARACTERS

The two basic functions which manipulate characters are:

CH$RCHAR
CH$WCHAR

The read function

CH$RCHAR(source-CS-pointer)

fetches the character referenced by the given pointer but leaves the CS-pointer itself unchanged.

The write function

CH$WCHAR(character, destination-CS-pointer)

stores the indicated character at the location referenced by the destination pointer, again leaving the pointer unchanged. For example:

```
    OWN
        CH1,                             !CHARACTER STORAGE
        CH2,
        CSP,                             !SOURCE POINTER
        DSP;                             !DESTINATION POINTER
```

```
CSP = CH$PTR(PLIT('EXAMPLE'));          !CSP IS A CS-POINTER TO
                                        ! THE STRING 'EXAMPLE'

DSP = CH$PTR(CH2);                      !DSP IS A CS-POINTER TO
                                        ! CH2

CH1 = CH$RCHAR(.CSP);                   !CSP STILL POINTS TO E
CH$WCHAR(.CH1,.DSP);                    !DSP STILL POINTS TO
                                        ! FIRST CHARACTER
                                        ! POSITION IN CH2
```

Note that the CS-pointers remain unchanged. That is, the value of
the pointer after executing the function is exactly the same as
the original pointer. Two such calls in succession would
therefore read (or write) the same character position twice.

These functions, with the appropriate extension, can also be
used to advance the CS-pointer during the read or write operation.
If the CS-pointer is to be advanced after the read or write
occurs, a "_A" is appended to the function name; if it is to be
advanced before performing the indicated operation, a "A_" is
prefixed to the function name after CH$. As these functions
update the CS-pointer, in addition to returning a character value,
they require the ADDRESS of the CS-pointer as their argument,
rather than its current value. Each function advances the
CS-pointer at the appropriate time and updates the pointer
location. For example, continuing from the above illustration:

```
    CH1 = CH$A_RCHAR(CSP);                  !CSP POINTS TO X

    CH$WCHAR_A(CH$RCHR(.CSP), DSP);         !DSP POINTS TO CH2+1

    CSP = CH$PLUS(.CSP,1);                  !CSP POINTS TO A

    CH1 = CH$RCHAR_A(CSP);                  !CSP POINTS TO M

    CH$WCHAR('I',.DSP);                     !DSP POINTS TO CH2+1

    CH$A_WCHAR(CH$RCHR(.CSP), DSP);         !DSP POINTS TO CH2+2

    CH1 = CH$RCHAR_A(CSP);                  !CSP POINTS TO P
```

Note that it is only the "_A" and "A_" forms of these character
functions which require the address of the CS-pointer. All other
character functions take the value of the CS-pointer.

FUNCTIONS WHICH INITIALIZE AND MOVE CHARACTER STRINGS

The following three functions move and initialize character strings:

    CH$MOVE
    CH$COPY
    CH$FILL

The function

    CH$MOVE (length, source-pointer, destination-pointer)

is used for copying a character sequence from one place to another.

The function

    CH$COPY (length-1, source-pointer-1,...,length-n,
        source-pointer-n, fill-character, destination-length,
        destination-pointer)

copies the concatenated character sequences to the given destination, filling unused positions with the fill character. Since this function accepts an arbitrary number of arguments, the last three arguments are always assumed to be the fill character, the destination length, and the destination pointer (in that order). The value of the CH$MOVE and CH$COPY functions is a CS-pointer referencing the destination character position following the last character moved.

The function

    CH$FILL (fill-character, length, destination-pointer)

is used for initializing a character sequence. The value of this function is a CS-pointer referencing the destination character position following the last character filled. To illustrate these functions, assume the following declarations (BLISS-32):

    OWN
        CSP,   !SOURCE POINTER
        DSP,   !DESTINATION POINTER
        CSP2,   !SECOND SOURCE POINTER
        WKSP:VECTOR[CH$ALLOCATION(100)]; !WORKING STORAGE

    BIND STP = UPLIT(
                    PLIT BYTE ('NOW '),
                    PLIT BYTE ('IS '),
                    PLIT BYTE ('THE '),
                    PLIT BYTE ('TIME '))

```
                          :VECTOR[4];
```

and the CS-pointer assignments:

```
     DSP = CH$PTR(WKSP[0]);               !CS-POINTER TO WORK
                                          ! AREA


     CSP = CH$PTR(.STP[3]);               !CS-POINTER TO T IN
                                          ! THE STRING 'TIME'
     CSP2 = CH$PTR(.STP[1]);              !CS-POINTER TO I IN
                                          ! THE STRING 'IS'
```

With these declarations and assignments, the expressions

```
     CH$FILL(%C' ', 100, .DSP);           !FILL WORK AREA WITH
                                          ! SPACES


     DSP = CH$MOVE(5,.CSP,.DSP);           !MOVE 'TIME ' TO
                                          ! WKSP[0-4]
                                          !DSP POINTS TO WKSP[5]


     DSP = CH$MOVE(3,.CSP2,.DSP);          !MOVE 'IS ' TO
                                          ! WKSP[5-7]
                                          !DSP POINTS TO WKSP[8]
```

and the expression

```
     DSP = CH$COPY(5, .CSP,
                   3, .CSP2,
                   %C' ', 100, .DSP);
```

both produce the same result in the work space WKSP and the string
"TIME IS ".

FUNCTIONS TO COMPARE CHARACTER SEQUENCES

     The following six functions are used to compare character
sequences:

```
     CH$GTR
     CH$GEQ
     CH$LSS
     CH$LEQ
     CH$EQL
     CH$NEQ
```

The general form of these functions is

```
    CH$xxx(length-1, CS-pointer-1,
            length-2, CS-pointer-2, fill-character)
```

where xxx indicates one of the relational operators (e.g., LSS,
LEQ, etc.) and the fill character defaults to 0. These functions
perform character-by-character comparisons, so that:

```
    IF .CS-pointer-1 xxx .CS-pointer-2
    THEN
        1
    ELSE
        0
```

For example, using the plit STP above,

```
    CH$NEQ(2, .CSP, 2, .CSP2);
```

returns the value 1 (for true). Alternatively,

```
    IF CH$NEQ(2, .CSP, 2, .CSP2)
    THEN
        TTY_PUT_QUO('TRUE')
    ELSE
        TTY_PUT_QUO('FALSE');
```

outputs to the terminal the string 'TRUE'.

FUNCTIONS USED TO SEARCH SEQUENCES

    The following four functions are provided for searching
character sequences:

```
    CH$FIND_SUB
    CH$FIND_CH
    CH$FIND_NOT_CH
    CH$FAIL
```


The function

```
    CH$FIND_SUB(length-1, CS-pointer-1, length-2, CS-pointer-2)
```

searches the sequence described by CS-pointer-1 and length-1 for
the first occurrence of an embedded subsequence precisely
identical to the sequence described by CS-pointer-2 and length-2.

The function

CH$FIND_CH(length, CS-pointer, character)

searches the sequence described by the CS-pointer and the given length for the first occurrence of the specified character.

Conversely, the function

CH$FIND_NOT_CH(length, CS-pointer, character)

searches for the first occurrence of a character not equal to the specified character. All three functions return either a CS-pointer referencing the matched subsequence/character or a null, if no match occurs.

The function

CH$FAIL(CS-pointer)

is used in conjunction with these three functions to examine the CS-pointer so that its value is:

```
IF CS-pointer EQL 0                      !EQUAL TO NULL POINTER
THEN
    1
ELSE
    0
```

The following example uses the plit STP (above) to demonstrate the CH$FAIL function:

```
!+
!  THIS LOOP SEARCHES FOR THE SUBSTRING 'ME'
!  IN EACH ENTRY IN THE PLIT STP
!-

INCR I FROM 0 TO 3 DO
    BEGIN
    LOCAL
        FOUND_PTR,
        LENGTH;

    MACRO
        PLIT_LENGTH(APLIT)=
            .((APLIT)-4) %;

    BIND
        ME = PLIT('ME');
```

```
          LITERAL
              ME_SIZE = %CHARCOUNT('ME');

          CSP = CH$PTR(.STP[.I]);
          LENGTH = PLIT_LENGTH(.STP[.I]) * 4;
          FOUND_PTR = CH$FIND_SUB(.LENGTH, .CSP, ME_SIZE,
                                       CH$PTR(ME));

          IF NOT CH$FAIL(.FOUND_PTR)
          THEN
              EXITLOOP;
          END;
```

This example returns a CS-pointer in CSP (on the fourth iteration) to the character M (matched in .STP[3]).

FUNCTIONS USED TO PERFORM CHARACTER TRANSLATION

The two functions which are used to perform character translation are:

    CH$TRANSTABLE
    CH$TRANSLATE

The first builds a translate table; the second performs a character by character translation.

The function

    CH$TRANSTABLE(translation)

builds a translation table at compile time. The translation syntax is the same as that for plits except that it must not include an allocation unit and it is limited to 256 characters.

The function

    CH$TRANSLATE(translate-table, length, source-pointer,
                 fill-character, length, destination-pointer)

performs the actual transformation. Each character in the source sequence is interpreted as an index in the translation table, and the corresponding character in the table (at that index) is moved to the destination string. If the destination sequence is longer than the source sequence, the destination is filled with the specified fill character. The value of the function is a CS-pointer referencing the character following the last character

moved into the destination sequence. For example, given the
translation table

```
BIND TBL = CH$TRANSTABLE(REP 36 OF (0), 38, 39,
                REP 8 OF (0), 37,
                0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                REP 7 OF (0),
                11, 12, 13, 14, 15, 16, 17, 18,
                19, 20, 21, 22, 23, 24, 25, 26,
                27, 28, 29, 30, 31, 32, 33, 34,
                35, 36,
                REP 37 OF (0));
```

the routine

```
ROUTINE ASCII_TO_RAD50(CSP,DSP): NOVALUE =
     CH$TRANSLATE(TBL, 1, .CSP, 0, 1, .DSP);
```

when passed the source pointer to a seven bit ASCII character and
a destination pointer, will write the corresponding RADIX-50
character representation to the destination address. Zero will be
written for any character not in the Radix-50 set.

Exercises

1. Write a translation table called NR_TBL which will translate a number from zero to nine into its ASCII equivalent, e.g., 0 to octal 60, 1 to octal 61, etc..

2. Given the address of an ASCIZ string (stored in ADDR), determine how many characters the first word (i.e., to the first space) contains. Assume the string starts with the first character of the word.

3. Determine the final result in TBL (list as a string) after excuting the following code:

```
OWN
    TBL:VECTOR[5],
    CSP,
    DSP,
    CHAR;

DSP = CH$PTR(TBL);
CSP = CH$PTR(PLIT('POINTER....'));

CH$COPY(7,CH$PLUS(.CSP,3),0,8,.DSP);
CH$FILL(%C'D',4,.DSP);

CHAR = CH$A_RCHAR(CSP);
CH$A_WCHAR(.CHAR,DSP);
CSP = CH$PLUS(.CSP,2);
CHAR = CH$RCHAR_A(CSP);
CH$A_WCHAR(.CHAR,.DSP);
CH$A_WCHAR(CH$A_RCHAR(CSP),DSP);
```

This page intentionally left blank.

Solutions

1) BIND
        NR_TBL = CH$TRANSTABLE(48,49,50,51,52,53,54,55,56,58);

2) OWN
        CSP,
        NR;

    CSP = CH$PTR(.ADDR);
    NR = CH$DIFF(.CSP,CH$FIND_CH(50,.CSP,%O'40'));

3) The ASCIZ string:

    DONE...

This page intentionally left blank.

Unit Test

Write a routine which, when passed the address of an ASCIZ
string (having less than 1000 characters) and the address of a
work area:

1) translates each character in the string into its Radix-50
    equivalent
2) stores the resulting translation into a work area
3) terminates the string with a null

The routine should use at least five different character handling
functions.

This page intentionally left blank.

DATA STRUCTURES
MODULE II-4

Course Map

```
                        ┌──────────┐
                        │UTILITIES │
                        └──────────┘


   ┌──────────┐      ┌──────────────┐     ┌──────────────┐
   │  TABLE   │      │ OPTIMIZATION │     │  DEBUGGING   │
   │ BUILDING │      └──────────────┘     │  TECHNIQUES  │
   └──────────┘             ▲             └──────────────┘
        ▲                   │                    ▲
        │                   │                    │
 ┌────────────┐             │             ┌──────────────┐
 │ ADDITIONAL │             │             │ CONDITIONAL  │
 │    DATA    │             │             │ COMPILATION  │
 │ STRUCTURES │             │             └──────────────┘
 └────────────┘             │                    ↗
        ▲                   │                   ╱
 ┌────────────┐      ┌────────────┐            ╱
 │   DATA     │◄─────│ CHARACTER  │───────────╱
 │ STRUCTURES │      │  HANDLING  │
 └────────────┘      │ FUNCTIONS  │
                     └────────────┘
                            ▲
                            │
                     ┌────────────┐
                     │  LEXICAL   │
                     │ FUNCTIONS  │
                     └────────────┘
                            ▲
                            │
                     ┌────────────┐
                     │  LANGUAGE  │
                     │  ABSTRACT  │
                     └────────────┘
```
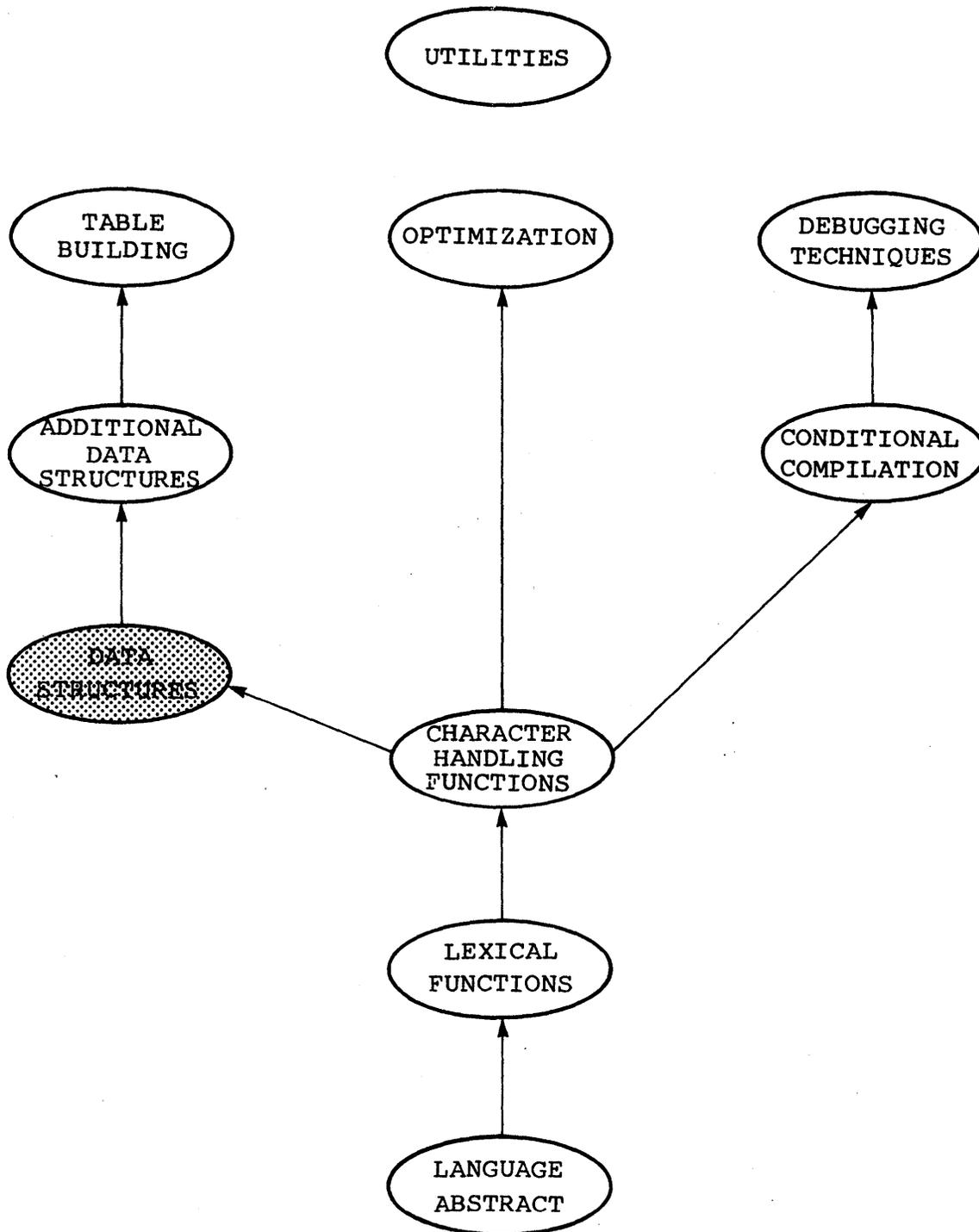
Introduction

This unit introduces the STRUCTURE declaration along with the related MAP declaration and REF attribute. The STRUCTURE declaration permits the user to define an arbitrary data structure; the MAP declaration associates a new accessing algorithm to existing storage; and the REF attribute simplifies indirect references to a structure.

Objectives

    Given an arbitrary data structure, be able to use the STRUCTURE declaration to create an accessing algorithm for that structure.

Sample Test Items

    Write a STRUCTURE declaration that implements a three dimensional array MATX3[row,column,plane], so that MATX3[1,2,0] references the second row, third column of the first plane. Show the declaration which allocates the storage and also a subsequent reference to the third column of the fifth row of the fourth plane.

Additional Resources

BLISS-32 Language Guide   Chapter 12: Data Structures
                                   Sections: 12.3 and 12.7

BLISS provides the capability to implement any arbitrary data structure. This is accomplished with the STRUCTURE declaration which permits the user to define an accessing algorithm for a given storage. As such, the accessing algorithm is simply an expression that returns an address based upon the access formal(s) for each reference. This declaration can also be used to compute the storage required for allocation of such a structure.

The predefined structures introduced earlier have been declared using this declaration mechanism. They are, in a sense, declared in an outer block of the user's program and, therefore, accessible from any block in the program. Since these structures should be familiar to most users, they are used as examples throughout this unit to illustrate the STRUCTURE declaration.

STRUCTURE FORMALS

The general form of the STRUCTURE declaration is:

```
STRUCTURE
    name[access-formals; allocation-formals] =
        [structure-size]
        (accessing-algorithm)<field-selector>;
```

As depicted in this declaration, there are two types of formal parameters: access formals, which are derived from the arguments used at the time of a structure reference, and allocation formals, which are constants derived from arguments at the time the structure is allocated. These formals are separated by a semicolon with access formals on the left and allocation formals on the right. For example,

```
STRUCTURE
    VECTOR[I;N] = ...
```

has one access formal, I, and one allocation formal, N. At the time of the storage declaration

```
OWN
    TABLE: VECTOR[10];
```

each occurrence of the allocation formal N is replaced by the allocation actual of 10. This is a one-time substitution. Similarily, the reference

```
TABLE[5]
```

causes each occurrence of the access formal I to be replaced by
the access actual of five in the accessing algorithm. The
distinction is that allocation formals, once declared, are
constants which do not change value; whereas access formals can
vary with each access reference to the structure.

The user has an option of assigning default values for
allocation formals. This is achieved by assigning a value to the
formal. For example,

```
STRUCTURE
    VECTOR[I;N,UNIT=4] = ...
```

designates UNIT as having a default value of four. Consequently,
if only one allocation actual were given in the storage
declaration, the default value four would be substituted for each
occurrence of UNIT.

STORAGE ALLOCATION

The next part of the STRUCTURE declaration is an optional
size component that is used to compute the amount, in bytes
(BLISS-16/32) or words (BLISS-36), of storage to be allocated.
For example (BLISS-32),

```
STRUCTURE
    VECTOR[I;N,UNIT=4] =
    [N*UNIT] ...
```

when invoked within the storage declaration

```
OWN
    TABLE: VECTOR[10,2];                    !VECTOR OF WORDS
```

allocates 20 bytes [10 * 2] or 10 words of storage with the
symbolic name TABLE; whereas

```
OWN
    TABLE: VECTOR[10];                      !VECTOR OF LONGWORDS
```

allocates 40 bytes [10 * 4] or 10 longwords. Note that in the
latter example, the default value for UNIT was used since only one
allocation actual was provided.

In a routine call, parameters are evaluated and stored at a
temporary location given by the applicable symbolic name of the
formal parameter. This is not the case in either macros or

structures where access actuals are substituted directly for the appropriate formal. As a consequence, the dot operator is not used to obtain the value of the access actual as is done in routines. That is, the following is erroneous,

    [.N * .UNIT] ...

if the value of N or UNIT is required.

Because a user may want to allocate structures in bytes, words, and longwords, the keywords BYTE, WORD, and LONG are predefined as literals having the values 1, 2, and 4 respectively when used as structure actuals or formals. (BLISS-16 does not recognize "LONG", while BLISS-36 recognizes only "WORD".) Consequently, the declarations

```
OWN
      TABLE: VECTOR[10,BYTE],            !VECTOR OF BYTES
      STORE: VECTOR[10,LONG];            !VECTOR OF LONGWORDS
```

are equivalent to:

```
OWN
      TABLE: VECTOR[10,1],               !VECTOR OF BYTES
      STORE: VECTOR[10,4];               !VECTOR OF LONGWORDS
```

ACCESSING ALGORITHM

The third part of a STRUCTURE declaration is the actual accessing algorithm or structure body. It is this algorithm that determines the address for each reference made to a specific logical element of the structure. For example, the STRUCTURE declaration for a vector of longwords (BLISS-32) could be written:

```
STRUCTURE
   L_VECTOR[I;N] =
       [N*4]
       (L_VECTOR + I * 4);
```

The subsequent storage declaration

```
OWN
      TABLE: L_VECTOR[100];
```

would then result in the allocation formal (in this example N) being replaced, producing:

```
STRUCTURE
    TABLE[I;N] =
        [100*4]
        (TABLE + I * 4);
```

This results in 100 longwords being allocated and in producing an
algorithm where only the access formals remain to be determined.
In a reference, such as

```
    TABLE[5];
```

the access actual (in this example, the argument five) replaces
the access formal (in this example the parameter I), and the
address resulting from the accessing algorithm is substituted in
place of the reference producing the address:

```
    (TABLE + 5 * 4);
```


FIELD SELECTOR

The final component of the STRUCTURE declaration is the field
selector. In BLISS the user has the capability to reference
and/or access any size field within a fullword. This is
accomplished with a field selector enclosed in "<>" of the form,

```
    <position, size, sign>
```

where position is an offset in bits from the address designated by
the symbolic name; size is the number of bits in the field; and
sign is the extension rule which applies to the field (zero
extension or sign extension). For this purpose, the keywords
SIGNED and UNSIGNED when used in structures are predefined
literals with the values 1 and 0 respectively. The sign default
is UNSIGNED. Note that BLISS-36 supports WORDs only; the
zero-extension vs. sign extension is meaningless for such
word-sized elements.

For example, assume a variable named FLAG contains the
following bit pattern as the result of the assignment:

```
    FLAG = %O'65';                                    !...0110101
```

To depict the resulting bit patterns, the rightmost digit (in this
example 1) represents bit zero; the digit to its left, bit one,
and so forth. The three dots at the left of the pattern indicate
that the remaining bits in the longword (through bit 31) are the

same as the leftmost digit shown (in this case 0).

Therefore, the reference

FLAG<4,3>   !...0110101
                                                                 ---

represents an unsigned field (by default) starting at the fifth
bit (bit four) and extending for three bits through the seventh
bit (bit six). As might be apparent, every reference to a name
not declared as a structure and not having an extension attribute
(SIGNED/UNSIGNED) or an allocation unit (BYTE, WORD, OR LONG)
contains an implied field selector. The default values are:

        BLISS-16:   <0,16,0>
        BLISS-32:   <0,32,0>
        BLISS-36:   <0,36,0>.

This means that the assignments (in, say, BLISS-36)

        FLAGS = .SAVE_FLAGS;

and

        FLAGS<0,36,0> = .SAVE_FLAGS<0,36,0>;

are equivalent. As FLAG now contains the value 53 (octal 65), the
assignment

        SAVE_FLAGS = .FLAG<4,3>;   !...011

sets SAVE_FLAGS to the value three. Note that fields are always
filled from right to left (left to right in BLISS-36) and that

    1) if the receiving field is larger than the sending
       field, the unspecified bits are set to zero; whereas

    2) if the receiving field is smaller than the sending
       field, the leftmost bits of the sending field are
       truncated.

The application of these rules is illustrated in the following
examples:

        SAVE_FLAGS = .FLAG<0,6>;   !...0110101
        SAVE_FLAGS<0,4> = 0;   !...0110000
        SAVE_FLAGS<0,5> = .FLAG<0,1>;   !...0100001

In actual practice, however, field selectors are only rarely used

in a direct assignment expression. They usually appear as the
last component of a STRUCTURE declaration where the same effect
can be achieved without cluttering in-line code.

VECTOR STRUCTURE

The BLISS-32 VECTOR structure is predefined as:

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

The definition of this structure permits a vector of signed or
unsigned bytes, words, or longwords. As examples, the declaration

```
OWN
    BYTE_TABLE: VECTOR[10,BYTE],        !UNSIGNED BYTE VECTOR
    WORD_TABLE: VECTOR[10,2,SIGNED],    !SIGNED WORD VECTOR
    LONG_TABLE: VECTOR[10];             !UNSIGNED LONGWORD
                                        ! VECTOR
```

allocates

```
    [10 * 1]                            !10 BYTES
    [10 * 2]                            !10 WORDS
    [10 * 4]                            !10 LONGWORDS
```

and defines the accessing algorithms:

```
    (BYTE_TABLE + I * 1)<0,8*1,0>       !BYTE FIELDS
    (WORD_TABLE + I * 2)<0,8*2,1>       !WORD FIELDS
    (LONG_TABLE + I * 4)<0,8*4,0>       !LONGWORD FIELDS
```


Compare this declaration to the BLISS-16 VECTOR declaration:

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

and the VECTOR declaration for BLISS-36:

```
STRUCTURE
    VECTOR[I;N] =
```

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

(Note that the VECTOR structure declaration for BLISS-36 is simplified due to the lack of a need to specify the Allocation Unit default value.)

BITVECTOR STRUCTURE

        The BITVECTOR structure is predefined in BLISS-32 as:

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

Note that storage is always allocated in bytes so that the declaration

```
OWN
     SWITCH_1: BITVECTOR[10];          !10 BIT VECTOR
     SWITCH_2: BITVECTOR[60];          !60 BIT VECTOR
```

allocates:

```
     [(10+7)/8]                        !2 BYTES OF STORAGE
     [(60+7)/8]                        !8 BYTES OF STORAGE
```

The accessing algorithm for this structure computes the field selector position offset from the beginning address. Consequently,

```
     SWITCH_1[5]
     SWITCH_2[45]
```

generates

```
     SWITCH_1<5,1,0>                   !5 IS THE SPECIFIED BIT
     SWITCH_2<45,1,0>                  !45 IS THE SPECIFIED BIT
```

and selects the fifth and forty-fifth bits respectively, counting from zero.

BLISS-16 declares the BITVECTOR structure as:

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

while BLISS-36 uses this declaration:

```
BITVECTOR[I;N] =
    [(N+35)/36]
    (BITVECTOR+((36^18+I)/36-1^18))
    <(36^18+I)MOD 36,1,0>;
```

BLOCK AND BLOCKVECTOR STRUCTURES

The BLISS-36 BLOCKVECTOR structure is predefined as:

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

As an illustration (using 36-bit architecture), consider the declaration

```
OWN
    B_BLOCKVEC: BLOCKVECTOR[10,5],      !5 WORDS PER
                                        ! BLOCK

    W_BLOCKVEC: BLOCKVECTOR[10,3],      !3 WORDS PER
                                        ! BLOCK

    L_BLOCKVEC: BLOCKVECTOR[10,2];      !2 WORDS PER
                                        ! BLOCK
```

which allocates

```
[10*5]                  !50 WORDS - 10 BLOCKS
[10*3]                  !30 WORDS - 10 BLOCKS
[10*2]                  !20 WORDS - 10 BLOCKS
```

and defines the accessing algorithms

```
(B_BLOCKVEC + O + I * 5)<P,S,E>
(W_BLOCKVEC + O + I * 3)<P,S,E>
(L_BLOCKVEC + O + I * 2)<P,S,E>
```

where the accessing formal I is the block number; O is the word

within the block; P is the field position; S is the field size; and E is the sign extension rule. A typical reference to this structure would be

        W_BLOCKVEC[5,2,18,18,0];

which, after replacing access formals in the algorithm with access actuals, determines the desired address and its field selector as shown below:

        (W_BLOCKVEC + 2 + 5 * 3)<18,18,0>;

In this example, it is the address of the lefthalf of the 18th word or logically to the user a data field in the righthalf of the third word of the sixth block.

        The BLISS-16 declaration for the BLOCKVECTOR structure is:

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

BLISS-32 predefines the BLOCKVECTOR structure as:

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

        The predefined BLOCK structures are identical to the BLOCKVECTOR structure without the access formal I and the allocation formal BS.

MAP

        Frequently, it is convenient to redefine an existing structure to have different properties. Savings in space, time, and readability can result. The MAP declaration provides this capability by permitting the user to associate a new accessing algorithm with an existing storage. As a declaration, MAP has the same scope rules as any other declaration. Therefore, the MAP declaration only applies within the block declared and all inner

blocks so long as the name of that storage is not redeclared or a
new accessing algorithm assigned to that storage.   Consider the
example (BLISS-36)

```
BEGIN
OWN
    SWITCH_TABLE: BITVECTOR[36];
...

    BEGIN
    MAP
        SWITCH_TABLE;

    SWITCH_TABLE = 0;
    END;
...

END;
```

which uses the MAP declaration to redefine SWITCH_TABLE from a
BITVECTOR to a scalar.   This enables SWITCH_TABLE to be set to
zero with a single assignment. After executing the assignment,
the block with the MAP declaration is exited, and the BITVECTOR
structure is again the accessing algorithm for SWITCH_TABLE.

Another example which uses the MAP declaration is the
BLISS-32 routine OUT_HEX below, which permits hexadecimal
characters, stored in a general purpose work area, to be output
without having to use shifts or field selectors explicitly.

```
    BEGIN
    LITERAL
        WK_SIZE = 10;

  OWN
        WK_AREA: VECTOR[WK_SIZE];
    ...

    ROUTINE OUT_HEX: NOVALUE =
        BEGIN
        STRUCTURE HEXVEC[I,J]=
            (HEXVEC + I)<J,4>;

        LITERAL
            FIRST = 0,
            SECOND = 4;

        MAP
            WK_AREA: HEXVEC;

        INCR COUNT FROM 0 TO (WK_SIZE*4-1) DO
            BEGIN
            IF .WK_AREA[.COUNT,FIRST] GTR 9
            THEN
                TTY_PUT_CHAR(.WK_AREA[.COUNT,FIRST] + %O'67')
            ELSE
                TTY_PUT_CHAR(.WK_AREA[.COUNT,FIRST] + %O'60');

            IF .WK_AREA[.COUNT,SECOND] GTR 9
            THEN
                TTY_PUT_CHAR(.WK_AREA[.COUNT,SECOND] + %O'67')
            ELSE
                TTY_PUT_CHAR(.WK_AREA[.COUNT,SECOND] + %O'60');

            TTY_PUT_CRLF();
            END;

        END;
    END;
```

Note that octal 60 is added to the contents of each byte to
produce the ASCII code required for output to the TTY.

In general, the use of field selectors should be restricted
to STRUCTURE declarations, as in the above example, since it makes
the resulting code easier to read, debug, and modify.

REF VECTOR

When using routines, it is common to pass the address of a
storage area as an argument of the routine call. In this instance
the location cannot be referenced directly by name but is pointed
to by the value of a variable. This unfortunately adds an
additional level of complexity by requiring an additional level of
indirection. The following routine, which exchanges the contents
of two vectors, illustrates the problem:

```
ROUTINE EXCH(VEC1,VEC2,SIZE): NOVALUE =
    BEGIN
    LOCAL
        TEMP;

    INCR COUNT FROM 0 TO (.SIZE - 1) DO
        BEGIN
        TEMP = .(.VEC1 + .COUNT);
        (.VEC1 + .COUNT) = .(.VEC2 + .COUNT);
        (.VEC2 + .COUNT) = .TEMP;
        END
    END;
```

This is more complicated than necessary. One solution is to
declare a new structure which returns the address of an element of
the vector. For example:

```
ROUTINE EXCH(VEC1,VEC2,SIZE): NOVALUE =
    BEGIN
    STRUCTURE
        PTR_VEC[C] =
            (.PTR_VEC + C);

    LOCAL
        TEMP;

    MAP
        VEC1: PTR_VEC,
        VEC2: PTR_VEC;

    INCR COUNT FROM 0 TO (.SIZE - 1) DO
        BEGIN
        TEMP = .VEC1[.COUNT];
        VEC1[.COUNT] = .VEC2[.COUNT];
        VEC2[.COUNT] = .TEMP;
        END;
    END;
```

This however requires a STRUCTURE declaration for a requirement
which is very common. Therefore, the REF attribute has been
provided. It enables a name, which contains a pointer to a
structure, to be used to reference the structure as if the name
were the structure itself, i.e., using only a single dot operator
instead of two. This is effectively equivalent to defining a new
structure definition. Using the REF attribute, the above example
can be rewritten as:

```
ROUTINE EXCH(VEC1,VEC2,SIZE): NOVALUE =
    BEGIN
    LOCAL
        TEMP;

    PVEC2: REF VECTOR;
    PVEC1: REF VECTOR;

    PVEC2 = .VEC2;
    PVEC1 = .VEC1;

    INCR COUNT FROM 0 TO (.SIZE - 1) DO
        BEGIN
        TEMP = .PVEC1[.COUNT];
        PVEC1[.COUNT] = .PVEC2[.COUNT];
        PVEC2[.COUNT] = .TEMP;
        END;
    END;
```

Note: The REF attribute may be used with any structure.

This page intentionally left blank.

Exercises

1.   Using the predefined BLISS-36 VECTOR structure definition
given in this unit, write a BLISS-36 STRUCTURE declaration named
VECTOR1 to create an accessing algorithm which starts at one
rather than zero (i.e., TABLE[1] references the first longword in
TABLE).

2.   Write a BLISS-36 STRUCTURE declaration named
MATX[row,column] that implements a two dimensional zero origin
array, so that MATX[0,1] is the first row, second column. Show
the initial declaration which allocates the storage and a
subsequent reference to the third column of the fifth row.

This page intentionally left blank.

Solutions

1)   STRUCTURE VECTOR1 [I;N]=
          [N]
          (VECTOR1 + (I - 1))<0,36>;

2]   STRUCTURE MATX [I,J;K,L]=
          [K*L]
          (MATX + (I*L + J))<0,36>;
      ...

      OWN
          X: MATX[10,10];   !100 WORDS
                                                   !10 ROWS, 10
                                                   ! COLUMNS

      ...

      X[4,2];

      Note: The array MATX starts at [0,0].

This page intentionally left blank.

Unit Test

Write a STRUCTURE declaration that implements a three dimensional array MATX3[row,column,plane], so that MATX3[1,2,0] references the second row, third column of the first plane. Show the initial storage declaration and a subsequent reference to the third column of the fifth row of the fourth plane.

This page intentionally left blank.

# ADDITIONAL DATA STRUCTURES

MODULE II-5

Course Map

```
                        ┌───────────┐
                        │ UTILITIES │
                        └───────────┘


   ┌───────────┐      ┌──────────────┐      ┌──────────────┐
   │   TABLE    │      │ OPTIMIZATION │      │  DEBUGGING   │
   │  BUILDING  │      └──────────────┘      │  TECHNIQUES  │
   └───────────┘              ▲             └──────────────┘
         ▲                    │                    ▲
         │                    │                    │
 ┌───────────────┐            │            ┌──────────────┐
 │  ADDITIONAL   │            │            │ CONDITIONAL  │
 │     DATA      │            │            │ COMPILATION  │
 │  STRUCTURES   │            │            └──────────────┘
 └───────────────┘            │                    ▲
         ▲                    │                    │
         │                    │                    │
   ┌───────────┐      ┌──────────────┐             │
   │   DATA     │◄─────│  CHARACTER   │─────────────┘
   │ STRUCTURES │      │  HANDLING    │
   └───────────┘      │  FUNCTIONS   │
                      └──────────────┘
                             ▲
                             │
                      ┌──────────────┐
                      │   LEXICAL     │
                      │  FUNCTIONS    │
                      └──────────────┘
                             ▲
                             │
                      ┌──────────────┐
                      │  LANGUAGE    │
                      │  ABSTRACT    │
                      └──────────────┘
```
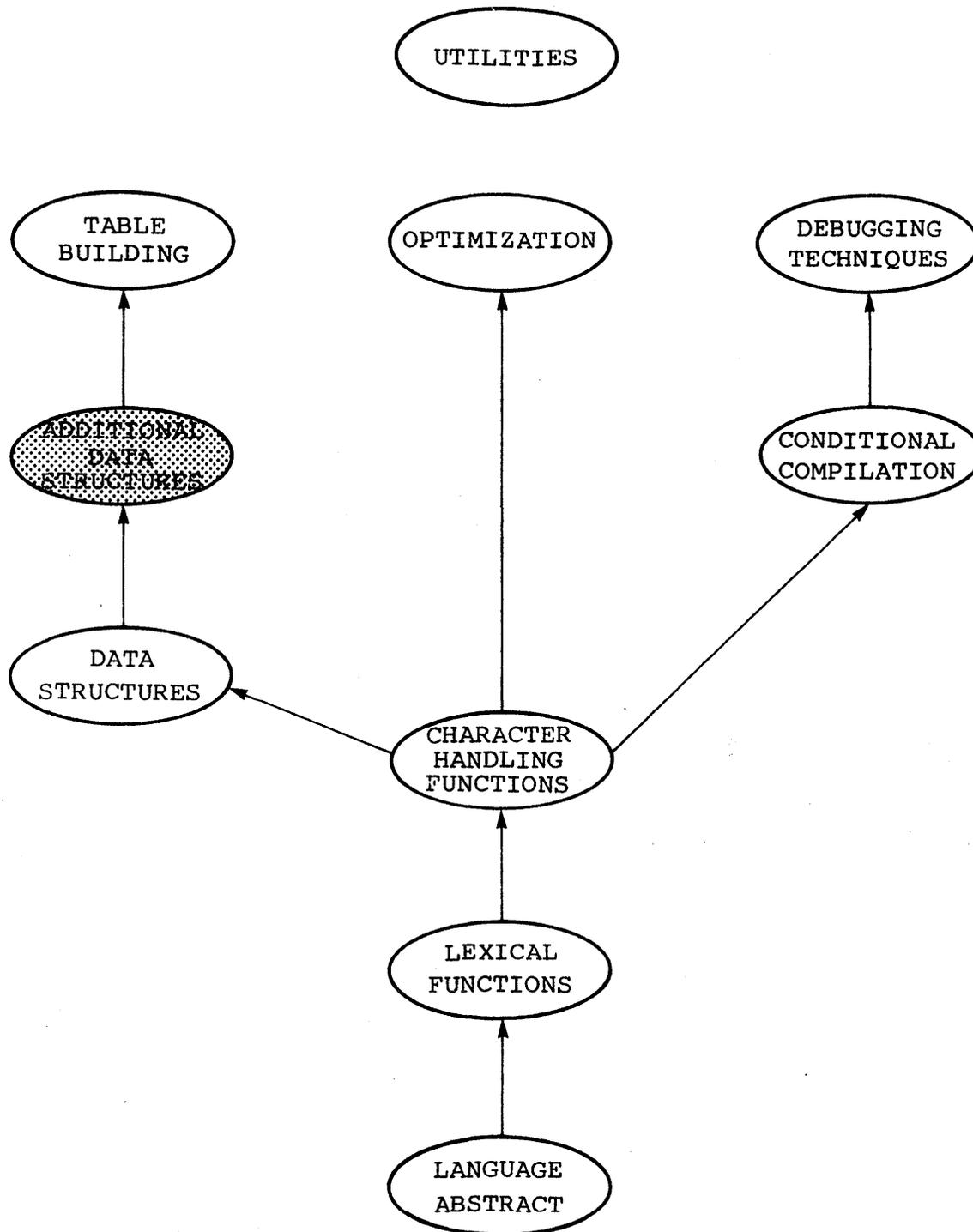
Introduction

This unit depicts several more examples of data structures and their associated field macros. These examples illustrate programming requirements for which the STRUCTURE declaration is appropriate. The structures presented in this unit involve bounds checking and linked lists.

──────── **Objectives** ────────

Given an arbitrary data structure, be able to implement this structure using the STRUCTURE declaration and appropriate field macros.

──────── **Sample Test Items** ────────

Given that a block contains the fields (for a 32-bit machine)
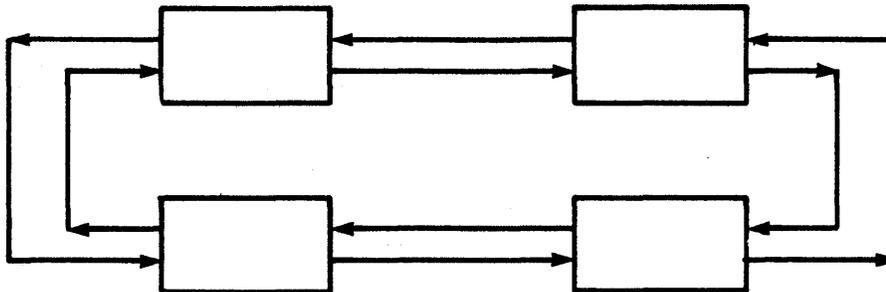
LEFT_LINK(16)	RIGHT_LINK(16)
TYPE(4)	DATA(28)

and that these blocks form a doubly linked ring,

write a STRUCTURE declaration named RING and the field macros necessary to implement this structure. Show the initial declaration allocating storage for the ring and a reference to the field TYPE made with a variable RING_PTR which is a REF to the ring.

──────── **Additional Resources** ────────

BLISS-32 Language Guide

Chapter 12: Data Structures
Section: 12.8

The predefined data structures are adequate for most programming requirements. However, it can be anticipated that special requirements will arise that necessitate the creation of new data structures. For this reason, BLISS provides the STRUCTURE declaration rather than attempting to define an all-purpose set of data structures which the user must force his program to use. Since the algorithm generated by a STRUCTURE declaration is substituted in-line for the actual structure reference, a STRUCTURE declaration should not be used to manipulate data or otherwise perform complex operations. A routine is the appropriate declaration when these operations are required.

This unit depicts examples which use the STRUCTURE declaration. These examples are intended to illustrate conditions when a new structure can be more appropriate than a predefined structure.

ERROR CHECKING

Error checking can be used advantageously during program debugging and for certain user application programs. As an example, consider the following BLISS-36 two dimensional array STRUCTURE declaration, which includes a simple bounds checking facility:

```
STRUCTURE
    ARRAY2[I,J;M,N] =
        [M*N]
        (IF I*J GTRU M*N
         THEN
             ERROR(ARRAY2,I,J)
         ELSE
             ARRAY2 + (I*M + J))<0,36>;
```

This structure, in addition to returning an address, also verifies that each reference is within the storage space allocated. It does not, however, ensure that each index is within bounds, only that their product does not exceed the storage space. (Note that the unsigned comparison (I*J GTRU M*N) precludes having to explicitly check that the product is less than zero, since a negative number would appear as a very large unsigned number which would always exceed the available storage.) If the product is within bounds, the normal address calculation is performed and returned. If the product of the access actuals (I*J) exceeds the declared storage (M*N), an error routine is invoked. In this example the routine ERROR is assumed to return a default address

for the illegal structure reference.   Since   the   value   returned
from   the   accessing   algorithm   will   be used as a reference, the
STRUCTURE   declaration   must   provide   a   default   address   for
out-of-bounds   conditions   to   preclude   terminating   the   program
because of an illegal address or, worse yet, an erroneous   address
which is still within the user's space.

A NON-CONTIGUOUS STRUCTURE

        The predefined structures declare storage that is   contiguous
in memory. This may, however, not be consistent with the logical
structure which is necessary for a particular programming problem.
For   example,   consider a program which requires a fixed number of
blocks, each consisting of   three   fullwords,   and   an   additional
block,   consisting of four fullwords, to store supplementary data.
The additional data may or may not be applicable to   each   of   the
fixed   blocks   required.   One possible solution is to allocate the
entire seven fullwords for   each   block.   This   would   of   course
necessitate   substantial storage that would be poorly utilized.   A
better solution is to obtain the   storage   for   the   supplementary
data from a common area on an as-needed basis.   The implementation
requires that the block of supplementary data   be   linked   to   the
appropriate   fixed   block.   This   is   depicted   in   the   following
diagram:

II-5-6
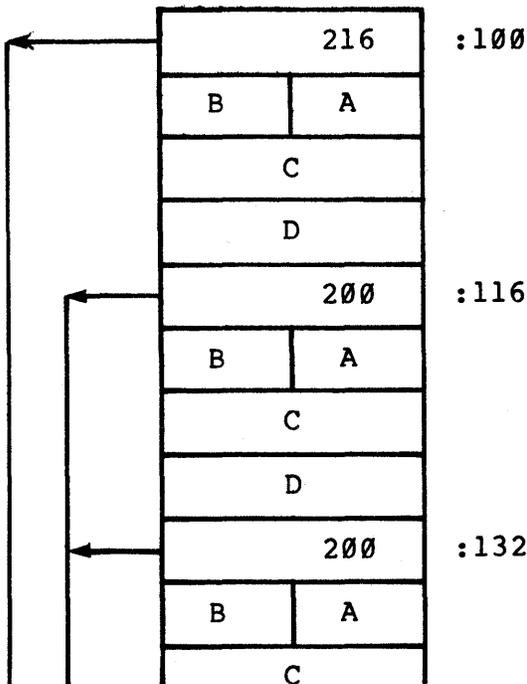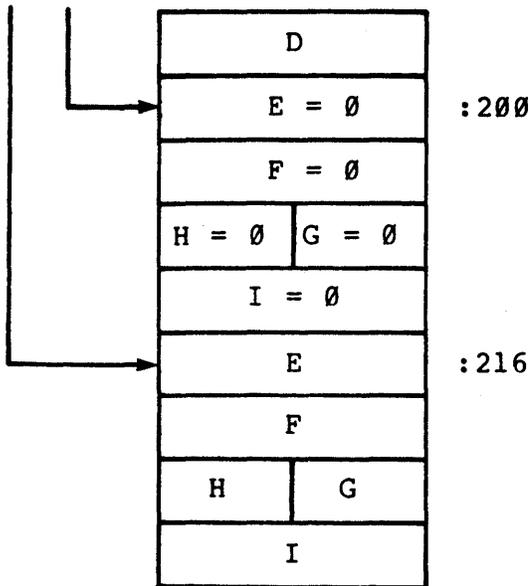
where A through D represent the data fields for the initial fixed blocks, and E through I represent the data fields for the supplementary data block. Note that the first fullword of each initial block is a pointer to the additional data block and that those with no additional data point for compatibility to a common block which is preset to zero.

The method used to allocate the storage would depend upon the initialization stragegy. Because this example deals with blocks, the predefined BLOCKVECTOR structure would probably be appropriate. For example,

```
    OWN
        DATA_BASE: BLOCKVECTOR[100,4];
```

declares 100 blocks having four fullwords each. A predetermined number of these blocks would be initialized with the first fullword set to the address of the first free block (in this example 200) which is preset to zero.

The accessing algorithm to implement the storage depicted above requires a STRUCTURE declaration. Since a STRUCTURE declaration and its associated field macros are necessary to understand the structure, they should be physically located together in the program to convey this relationship. For example (BLISS-32):

```
STRUCTURE
!+
!  B = BLOCK NUMBER, O = LONGWORD OFFSET,
!  P = FIELD POSITION, S = FIELD SIZE,
!  E = SIGN EXTENSION, I = INDIRECT ADDRESS
!-
    BLOCK_LINK[B,O,P,S,E,I] =
        (IF I EQL 0
            THEN
                (BLOCK_LINK + B*16 + O*4)
            ELSE
                (.BLOCK_LINK + B*16 + O*4)
            )<P,S,E>;

MACRO
!+
!  FIELD DEFINITIONS FOR BLOCK_LINK STRUCTURE
!  FOR O,P,S,E,I RESPECTIVELY
!-
    A = 1,0,16,0,0 %,
    B = 1,16,16,0,0 %,
    C = 2,0,32,0,0 %,
    D = 3,0,32,0,0 %,
    E = 0,0,32,1,1 %,
    F = 1,0,32,1,1 %,
    G = 2,0,16,1,1 %,
    H = 2,16,16,1,1 %,
    I = 3,0,32,1,1 %;
```

This STRUCTURE declaration implements the structure discussed
above.   Note that the BLOCK_LINK structure contains no allocation
formals. This structure is used to access storage only and
allocates no storage. As written, this structure can be
associated to the previously allocated storage DATA_BASE using the
declaration

```
    MAP
        DATA_BASE: BLOCK_LINK;
```

so that subsequent references to the structure would have the
form:

```
    DATA_BASE[block-number, macro A-I]
```

The first entry is the desired block number, and the second entry
is one of the macros A through I defined above. Explicitly,

```
    DATA_BASE[2,G]
```
references the data field G in the third block.   Although the

physical block containing this supplementary data may be any block
in the vector, it is associated to block three through the
BLOCK_LINK structure.

The REF attribute could have been used instead of the MAP
declaration. The choice would depend to a large extent upon the
manipulation requirements of the program. With the REF attribute,
the block number is not necessary, but the address of the desired
block must be maintained. An example using REF is provided below.

TREE STRUCTURE

The structure illustrated above can also be applied to build
a tree structure. Assume a block configuration of,
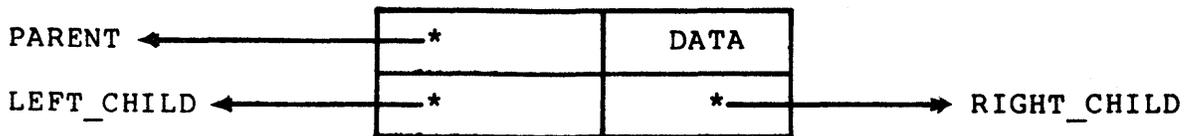
where the pointers for this discussion are offsets into the
storage vector rather than an actual address. As such, this
configuration can be used to define a binary tree (each parent
having at the most two children) as shown below:
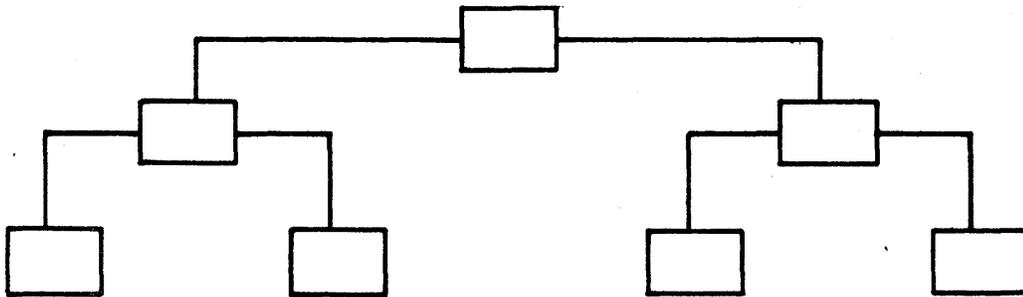
The declaration to allocate this storage can either be part of the
structure declaration or independent of it, depending upon the
initialization requirements. For example, the following STRUCTURE
declaration, and its corresponding field macros, implements the
algorithm for a tree but allocates no storage for that structure.
(The declaration is patterned for a 32-bit word length.):

```
STRUCTURE
!+
!  I = INDIRECT ADDRESS,           O = FULLWORD OFFSET,
!  P1 = INDIRECT FIELD POSITION,  S1 = INDIRECT FIELD SIZE
!  E1 = INDIRECT FIELD SIGN,
!  P = FIELD POSITION,             S = FIELD SIZE
!-
     TREE[I,O,P1,S1,E1,P,S] =
          (IF I EQL 0
               THEN
                    .(TREE + O*4)<P1,S1,E1> + TREE
               ELSE
                    TREE
               )<P,S>;

MACRO
!+
!  FIELD DEFINITION FOR TREE STRUCTURE
!  FOR I,O,P1,S1,E1,P,S RESPECTIVELY
!-
     PARENT = 0,0,16,16,1,0,32 %,
     DATA   = 1,0,0,0,0,0,16  %,
     LEFT_CHILD = 0,1,16,16,0,0,32 %,
     RIGHT_CHILD = 0,1,0,16,0,0,32 %;
```

Because this structure uses offsets instead of addresses, the algorithm must add the contents of the field to the base address of the storage area. The TREE structure, defined above, can be referenced by establishing a pointer with the declaration

```
OWN
     NODE: REF TREE;
```

where subsequent references such as

```
NODE = NODE[PARENT]
TEMP = .NODE[DATA]
```

would conceptually be replaced by the algorithms:

```
NODE = (.NODE[PARENT]<16,16> + NODE[PARENT])<0,32>
TEMP = .NODE[DATA]<0,16>
```

The first example assigns to NODE the address of the parent node (an offset in fullwords plus the current address in the TREE); whereas the second example assigns to TEMP the data for that node.

TREE TRANSVERSAL

To demonstrate the use of the TREE structure defined above, consider a tree constructed from parsing the equation

$$(5 - .B) = (.A + 7);$$

as depicted below:
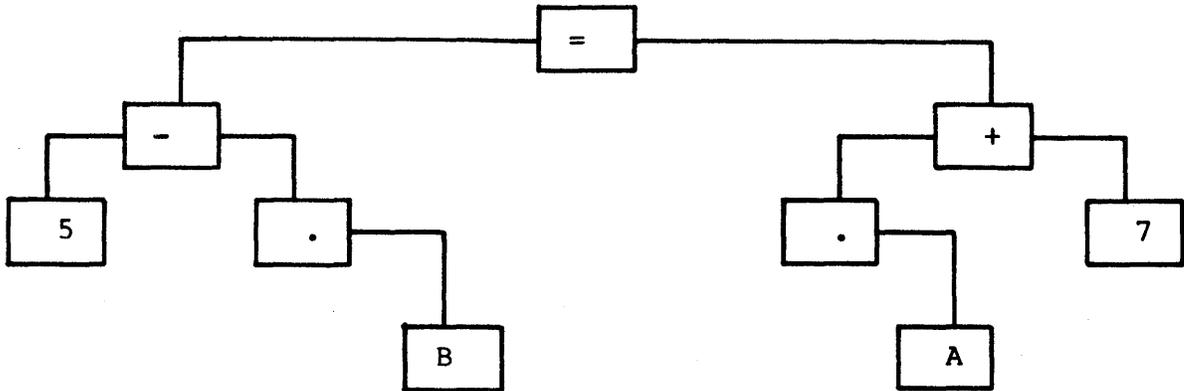
Assuming that the data is stored as ASCII characters in the data field, the following recursive routine could be used to perform an inorder (left-child, parent, right-child) transversal of any binary tree constructed with similar fields and output the contents of the data fields:

```
ROUTINE OUT_TREE(TOP_NODE): NOVALUE=
    BEGIN
    LOCAL
        NODE: REF TREE;
    NODE = .TOP_NODE;
    IF NODE[LEFT_CHILD] NEQ .NODE
    THEN
        OUT_TREE(NODE[LEFT_CHILD]);
    TTY_PUT_CHAR(.NODE[DATA]);
    IF NODE[RIGHT_CHILD] NEQ .NODE
    THEN
        OUT_TREE(NODE[RIGHT_CHILD]);
    END;
```

The routine OUT_TREE is recursive and continues to call itself so long as a left- or right-child exists. That is, so long as a left- or right-child field contains an offset other than zero. The routine searches first for a left-child; therefore, the algorithm prints the data inorder: left-child, parent, right-child until the entire tree has been transversed.

As mentioned, the fields in the above example contain offsets which must be added to the current base address; therefore, the parent field must be signed. In fact, had larger fields been allocated initially, they could have contained the actual addresses.

Exercises

1. Create a BLISS-36 structure named VECTOR_BCK that has a normal predefined VECTOR accessing algorithm but also does bounds checking on the access actual used in the reference. Call a routine named ERROR if the access actual is not in bounds and pass the reference actual as a parameter. You can assume that the routine ERROR returns a default address.

2. Same as Exercise 1) above except this exercise assumes ERROR does not return a default address. Make the first fullword of the storage area the default address (i.e., return the address of the first fullword of the array VECTOR_BCK[0] if a referencing error occurs).

This page intentionally left blank.

Solutions

1)   STRUCTURE
         VECTOR_BCK[I;K] =
             [K]
               (IF I LSS 0 OR I GTR (K-1)
               THEN
                     ERROR(VECTOR_BCK,I)
               ELSE
                     VECTOR_BCK + I)
                         <0,36>;

2)   STRUCTURE
         VECTOR_BCK[I;K] =
             [K]
             (LOCAL
                     TEMP;

             TEMP = I;

             IF I LSS 0 OR I GTR K-1)
             THEN
                     BEGIN
                     ERROR(VECTOR_BCK,I);
                     TEMP = 0;
                     END;

             VECTOR_BCK + .TEMP)
                 <0,36>;

This page intentionally left blank.

Unit Test

Given that a block contains the fields (32-bit machine)

LEFT_LINK(16)	RIGHT_LINK(16)	
TYPE(4)	DATA(28)	
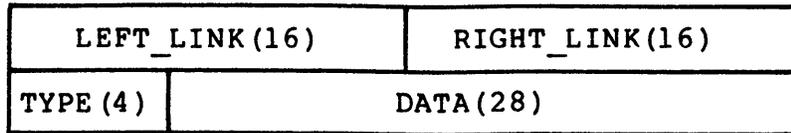
and that these blocks form a doubly linked ring,
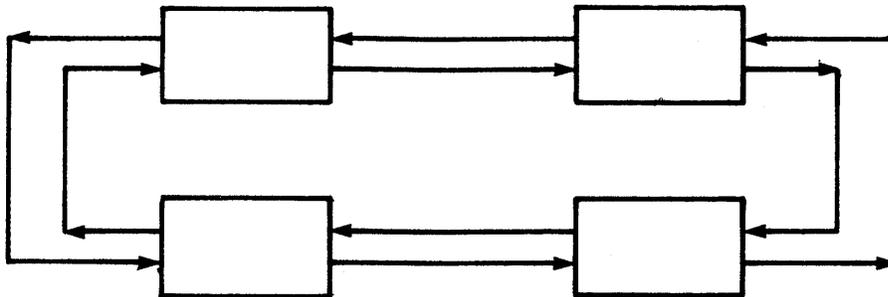
write a BLISS-32 structure declaration named RING and the field macros necessary to implement this structure. For purposes of accessing this structure declare RING_PTR as a REF to the ring. Show the initial declaration which allocates the storage for the ring and a reference which accesses the field TYPE. Assume that LEFT-LINK and RIGHT-LINK contain offsets from the address of the current block to the address of the desired block.

This page intentionally left blank.

TABLE BUILDING
MODULE II-6

Course Map

```
                    ( UTILITIES )



   (  TABLE   )      ( OPTIMIZATION )        ( DEBUGGING  )
   ( BUILDING )                              ( TECHNIQUES )
       ↑                    ↑                      ↑
       |                    |                      |
  ( ADDITIONAL )                           ( CONDITIONAL  )
  (   DATA     )                           ( COMPILATION  )
  ( STRUCTURES )                                   ↑
       ↑                    |                      |
       |                    |                      |
   (   DATA    ) ←   ( CHARACTER  )  →
   ( STRUCTURES)     (  HANDLING  )
                     ( FUNCTIONS  )
                           ↑
                           |
                     (  LEXICAL   )
                     ( FUNCTIONS  )
                           ↑
                           |
                     ( LANGUAGE  )
                     ( ABSTRACT  )
```
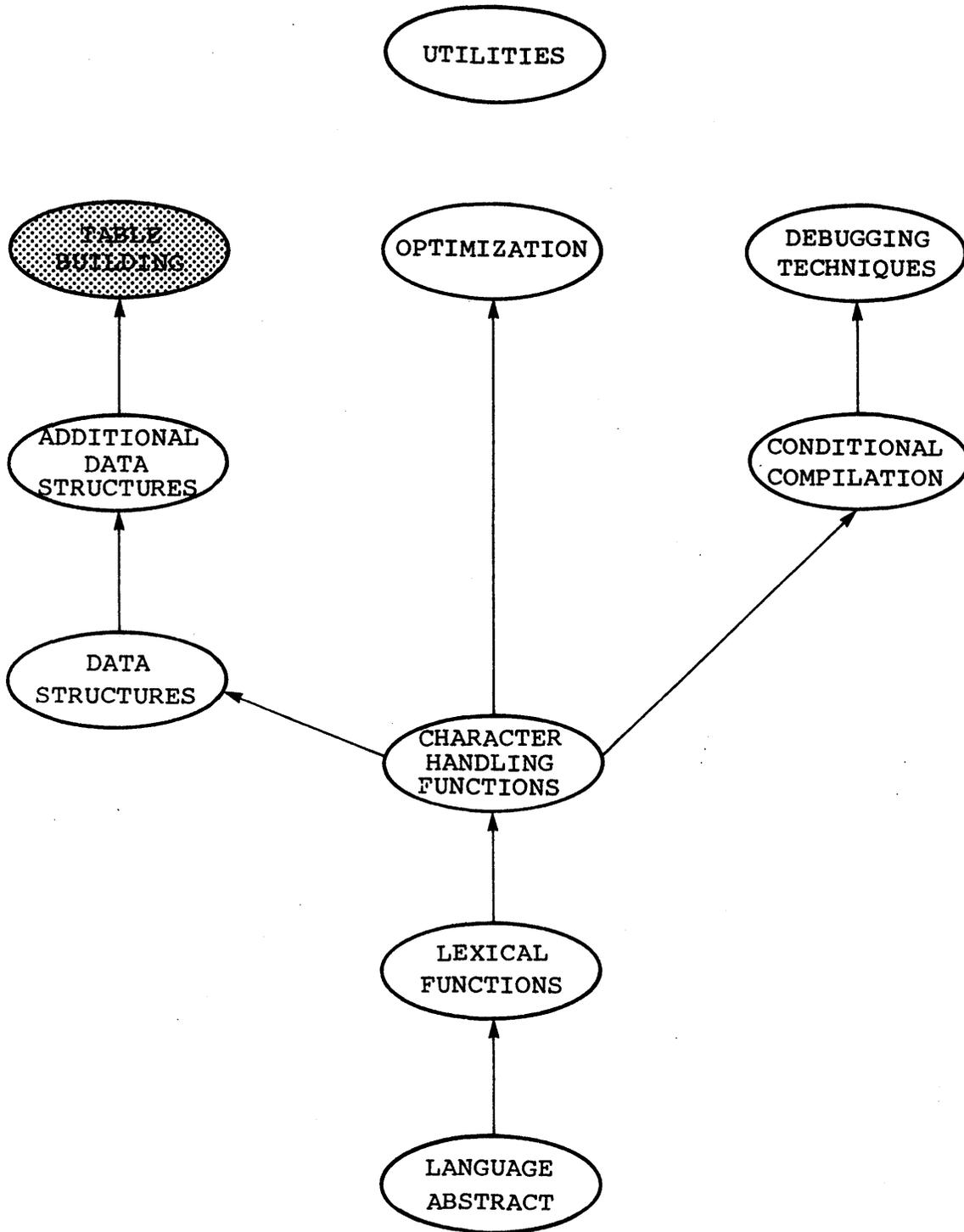
Introduction

Initialization, which is the final step in establishing a data structure, is discussed in this unit. Two methods of initializing a table are depicted: the INITIAL attribute for storage, which may subsequently be modified, and the PLIT declaration for applications where storage is not subject to change.

Objectives

Given a data structure and its required initial value, be able to initialize the given structure using the INITIAL attribute or the PLIT declaration as appropriate.

Sample Test Items

Given that a BLOCKVECTOR named TBL consists of six blocks, each having three fullwords, where the first three blocks (named MASK_BLK) have the logical fields (for 32-bit architecture)

PTR (32)		MASK_BLK:
S (16)	R (16)
MASK (32)	
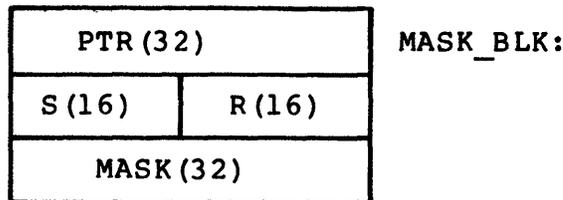
and the last three blocks (named DATA_BLK) have logical fields,
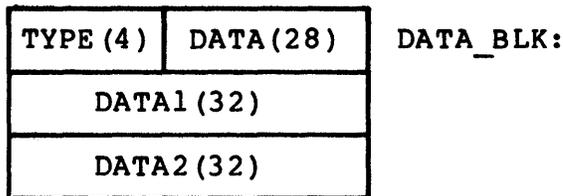
TYPE (4)	DATA (28)	DATA_BLK:
DATA1 (32)	
DATA2 (32)	

write the macros and declarations necessary to allocate and initialize TBL, so that:

1) MASK_BLK contains an offset of three (points to the fourth block) in PTR; zeroes in S; the address of the symbolic names A1 through A3, respectively, in R; and the value -1 in MASK

2) DATA_BLK contains the value nine in TPYE and the value zero in DATA, DATA1, and DATA2

There are three distinct steps in establishing most data structures:

1) defining an accessing algorithm
2) establishing field macros
3) initializing the storage

The process of defining the accessing algorithm has been discussed in the two previous prerequisite units. This process includes either using predefined structures or creating new structures with the STRUCTURE declaration. In general, the predefined structures are adequate for most applications and their use is encouraged. When a predefined structure would suffice, creating a new structure could prove to be inefficient with respect to execution time and could result in increased debugging time. It is, of course, not recommended that an algorithm be manipulated to fit a predefined structure when a new structure is clearly necessary. It is suggested that the data structure is an essential part of an efficient algorithm and should, therefore, be given considerable thought.

Field macros are also an essential part of a data structure. Field macros help to improve readability and documentation by providing easily recognized mnemonics, simplifing coding through textual substitution, and making the program easier to modify and debug by isolating the code.

The final phase of implementing a given data structure is its initialization. The particular method of initialization is contingent on the data structure's size, complexity, and subsequent utilization. Two methods will be illustrated: initializing with the INITIAL attribute and with the PLIT declaration.

BASIC TECHNIQUE

The basic technique in table building is to fill up the table one fullword at a time, by shifting and ORing the values of individual fields as necessary. For example, the expression

$$1 \char94 28 \quad OR \quad 100 \char94 12 \quad OR \quad 'S' \char94 4 \quad OR \quad 3$$

would generate one fullword, consisting of four separate fields organized as follows (36-bit machine):

A(8)	B(16)	C(8)	D(4)
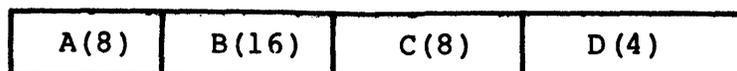
The shift factor is computed as the sum of the sizes of all the fields to the right of the one in the current fullword.

INITIALIZATION WITH THE INITIAL ATTRIBUTE

The INITIAL attribute permits the user to initialize permanent storage to any given value at the time it is allocated. For example,

```
OWN
    TABLE: VECTOR[10]
            INITIAL(REP 10 OF (0));
```

sets each of the 10 locations in the vector TABLE to zero, while

```
OWN
    B_TABLE: VECTOR[4]
            INITIAL('EXAMPLE');
```

initializes the vector B_TABLE to 'EXAMPLE'. However, the initialization becomes increasingly more complicated as fields within each word unit require individual values. This is especially true with blocks or vectors of blocks. Consider, for example, the diagram

A(8)	B(16)	C(8)	D(4)
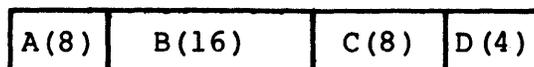
which is logically divided into fields A through D. The number of bits in each field is indicated by the numbers in parentheses. The initialization of these fields to 1, 100, 'S' and 3, respectively, could be achieved with the INITIAL attribute as in the following declaration:

```
OWN
    X: INITIAL(1^28 OR 100^12 OR %C'S'^4 OR 3);
```

However, this form is awkward and is also not particularly descriptive. Furthermore, this is a relatively simple example having only one fullword. A vector of 25 fullwords would be tedious, more prone to errors, and a considerable headache if one or more of the fields required modification. One solution to this

problem is to use a macro to initialize the fields. For example,

```
    MACRO
        WORD_INIT(A,B,C,D)=
            INITIAL((A)^28 OR (B)^12 OR (C)^4 OR (D)) %,
        VEC_INIT(A,B,C,D)=
            ((A)^28 OR (B)^12 OR (C)^4 OR (D)) %;
```

could be used to initial a fullword as in the example above,

```
    OWN
        X: FULL_INIT(1,100,'S',3);
```

or to initialize a vector of these fullwords:

```
    OWN
        TABLE: VECTOR[10]
            INITIAL(VEC_INIT(1,100,%C'A',3),
                    VEC_INIT(2,101,%C'B',4),
                    ...

                    VEC_INIT(10,110,%C'J',12));
```

The procedure is easily adapted to BLOCKS, BITVECTORS, or any other structure. In addition, initialization macros offer the same advantages discussed for field macros above and should therefore be an essential part of any initialization. As declared above, the initialized values could easily be modified.

Where the storage once initialized is not subject to further modification, a PLIT declaration can be used to advantage.

INITIALIZATION WITH PLITS

As the following example illustrates, plits can be used to build a table much in the same manner as was achieved with the OWN declarations above:

```
    BIND
        TABLE = UPLIT(
                    VEC_INIT(1,100,%C'A',3),
                    VEC_INIT(2,110,%C'B',4),
                    ...

                    VEC_INIT(10,110,%C'J',12))
                        : VECTOR[10];
```

Note that both the OWN and PLIT declarations use initialization macros and also that both can subsequently be referenced with

    TABLE[...]

if the plit is used in conjunction with the BIND declaration as above. Note that UPLIT is the preferred declaration unless the count is required. AN INITIALIZATION EXAMPLE

The technique for multiword tables is simply a repetition of the packing methods discussed above, with commas separating the values for each word. Consider initializing a symbol table having three fullwords per block, where each block is logically defined as:

N1(32)	
T(1Ø)	N2(22)
A(32)	
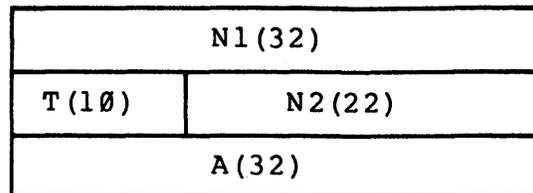
In the above diagram, N1 is the Radix-5Ø representation of the first six letters in the name; N2 is the Radix-5Ø representation of the last four letters, T is a type field giving information about the name; and A is the address of the routine which processes that name. To keep this example simple, only four names of this table will be initialized. The following macro initializes the block:

```
    MACRO
        INIT_TBL(A,B,C,D) =
            %RAD5Ø_11(%STRING(A)),
            (C)^22 OR %RAD5Ø_11(%STRING(B)),
            (D) %;
```

The storage for these blocks can be allocated and initialized in a plit since the symbol table will not be subsequently modified. For example,

```
    BIND
        SYM_TABLE = UPLIT(
                        INIT_TBL('INCR',Ø,2,SINCR),
                        INIT_TBL('IF',Ø,2,SIF),
                        INIT_TBL('ROUTIN',%C'E',1,SROUTINE),
                        INIT_TBL('SELECT','ONE',Ø,SSELECT))
                            :BLOCKVECTOR[4,3];
```

initializes storage as depicted below,

```
┌─────────────────────────────────┐
│              INCR               │
├─────┬───────────────────────────┤
│  2  │                        Ø  │
├─────┴───────────────────────────┤
│                            *────┼──────────► ROUTINE SINCR
├─────────────────────────────────┤
│               IF                │
├─────┬───────────────────────────┤
│  2  │                        Ø  │
├─────┴───────────────────────────┤
│                            *────┼──────────► ROUTINE SIF
├─────────────────────────────────┤
│             ROUTIN              │
├─────┬───────────────────────────┤
│  1  │                        E  │
├─────┴───────────────────────────┤
│                            *────┼──────────► ROUTINE SROUTINE
├─────────────────────────────────┤
│             SELECT              │
├─────┬───────────────────────────┤
│  Ø  │           ONE             │
├─────┴───────────────────────────┤
│                            *────┼──────────► ROUTINE SSELECT
└─────────────────────────────────┘
```
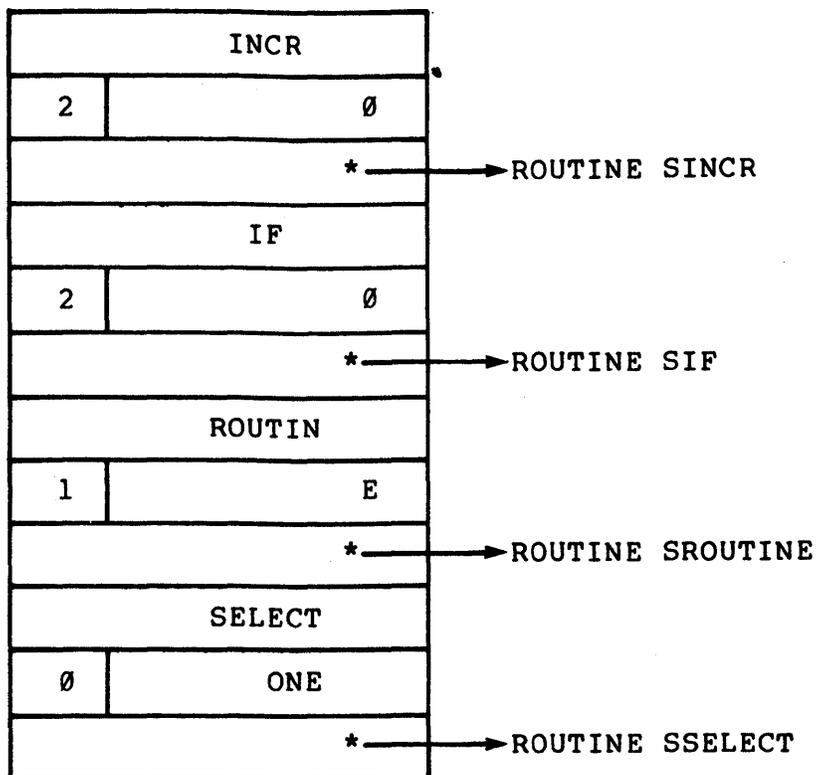
where the names would be in Radix-5Ø. An arrow (──►) indicates a
pointer to the specified data.

This page is for notes.

Exercises

1. Declare a macro named INIT_VAL which initializes a WORD
(BLISS-16/32) with the logical fields:

XSIGN(1)	EXPONENT(6)	MANTISSA(8)	MSIGN(1)

where the numbers in parentheses represent the number of bits in
each field. Show the macros and the actual declaration necessary
to initialize a word named REAL declared in an OWN declaration so
that:

```
XSIGN = 1
EXPONENT = 240
MANTISSA = 128
MSIGN = 0
```

2.   Using the format in Exercise 1) above, declare a macro to
initial a plit bound to the name REAL_TBL consisting of 10 words.
Show the first and 10th entries (use "..." for the missing data in
between) using any data.

3.   Construct a BLISS-36 macro named INIT_BLK to initialize a
block with the fields:

L_ADDR(18)		R_ADDR(18)	
TYPE(6)	DATA(18)		PTR(12)

Show the initialization of a BLOCK named HEAD so that:

```
L_ADDR = 100
R_ADDR = 200
TYPE = 1
DATA = 'A'
PTR = 100
```

This page intentionally left blank.

Solutions

```
1)    MACRO
          INIT_VAL(A,B,C,D) =
              INITIAL((A)^15 OR (B)^9 OR (C)^1 OR (D)) %;

      OWN
          REAL: WORD
              INIT_VAL(1,240,128,0);


2)    MACRO
          INIT_VAL(A,B,C,D) =
              (A)^15 OR (B)^9 OR (C)^1 OR (D) %;

      BIND
          REAL_TBL = UPLIT(
                          INIT_VAL(1,240,128,0),
                          ...

                          INIT_VAL(10,250,138,10))
                              :VECTOR[10,WORD];


3)    MACRO
          INIT_BLK(A,B,C,D,E) =
              INITIAL((A)^18 OR (B),
              (C)^30 OR (D)^12 OR (E)) %;

      OWN
          HEAD: BLOCK[2]
              INIT_BLK(100,200,1,%C'A',100)
```

This page intentionally left blank.

Unit Test

Given that a blockvector named TBL consists of six blocks, each having three fullwords, where the first three blocks (named MASK_BLK) have the logical fields (for a 36-bit machine)

```
+-----------------------+
|       PTR(36)     .   |   MASK_BLK:
+-----------+-----------+
|  S(18)    |   R(18)   |
+-----------+-----------+
|        MASK(36)       |
+-----------------------+
```
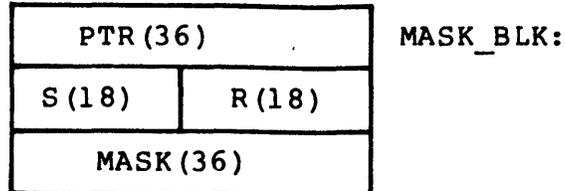
and the last three blocks (named DATA_BLK) have the logical fields,
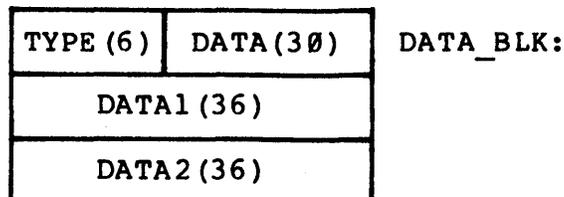
```
+---------+-------------+
| TYPE(6) |  DATA(30)   |   DATA_BLK:
+---------+-------------+
|        DATA1(36)      |
+-----------------------+
|        DATA2(36)      |
+-----------------------+
```

write the macros and declarations necessary to allocate and initialize TBL, so that:

1) MASK_BLK contains an offset of three (points to the fourth block) in PTR; zeroes in S; the address of the symbolic names A1 through A3, respectively, in R; and the value -1 in MASK

2) DATA_BLK contains the value nine in TYPE and the value zero in DATA, DATA1, and DATA2

This page intentionally left blank.

CONDITIONAL COMPILATION

MODULE II-7

Course Map

```
                        ( UTILITIES )


   (   TABLE    )      ( OPTIMIZATION )      (  DEBUGGING  )
   (  BUILDING  )                            ( TECHNIQUES  )
        ↑                     ↑                     ↑
   ( ADDITIONAL )                            ( CONDITIONAL )
   (    DATA    )                            ( COMPILATION )
   ( STRUCTURES )
        ↑
   (    DATA    ) ←      ( CHARACTER  )  →
   ( STRUCTURES )        (  HANDLING  )
                         ( FUNCTIONS  )
                              ↑
                         (  LEXICAL   )
                         ( FUNCTIONS  )
                              ↑
                         (  LANGUAGE  )
                         (  ABSTRACT  )
```
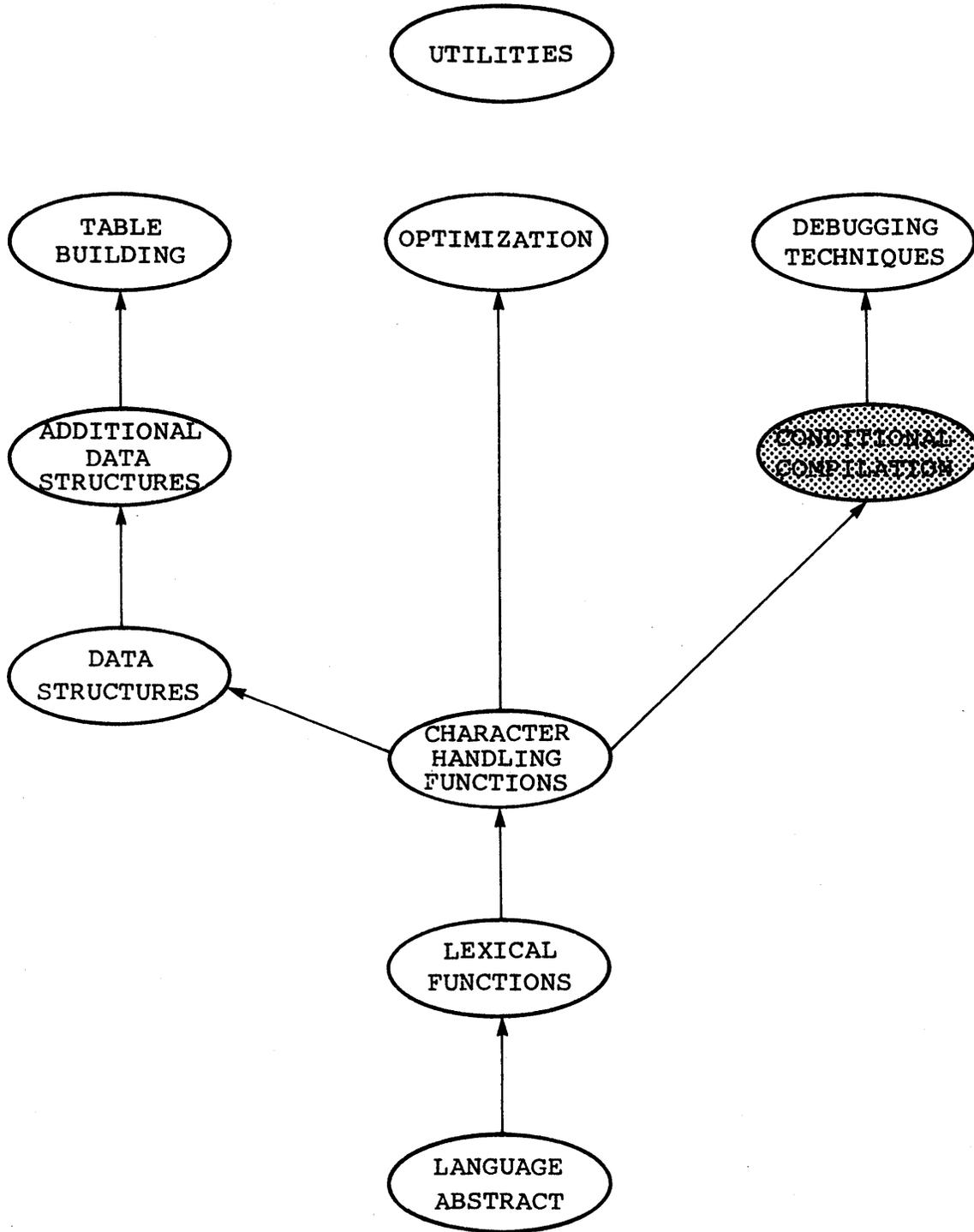
Introduction

This unit discusses the conditional compilation mechanism available in BLISS along with various lexical functions which are useful in combination with this capability.

─────── **Objectives** ───────

Given a programming problem, be able to develop a solution which correctly uses the conditional compilation mechanism and related lexical functions.

─────── **Sample Test Items** ───────

Write a macro named INIT that uses conditional compilation to initialize a vector depending upon the compiler used, so that:

    BLISS 36 -
        Fields: A = <0,18>
                  B = <18,18>
    BLISS 32 -
        Fields: A = <0,8>
                  B = <8,8>
                  C = <16,16>

In addition, the solution should terminate the compilation with an error diagnostic if:

* required arguments are missing
* the macro is called more than some arbitrary number of times (use two compile-time constants named LIMIT and COUNT)

Assume that COUNT is initially set to zero; LIMIT is set to an arbitrary value at the beginning of each vector initialization; and COUNT is incremented by one on each call.

─────── **Additional Resources** ───────

None

Conditional compilation provides a means to select, during compilation, one of two alternative sources of input text. The choice is determined by testing a compile-time expression. The general form is:

```
%IF compile-time expression
%THEN
     source-code sequence one
%ELSE
     source-code sequence two
%FI
```

The compile-time expression is evaluated, and the value of the low-order bit is used to select the appropriate source code sequence. A one selects the %THEN part whereas a zero selects the %ELSE part. The sequence selected is incorporated into the program, while the sequence not selected is discarded. The %ELSE part is optional and, if not present, a null lexeme is assumed. Note that a conditional compilation begins with a %IF and ends with a %FI. As a consequence, a semicolon is not required. An example of a conditional compilation:

```
%IF COND_FLAG
%THEN
     MACRO
         FIELD = 0,8 %;
%ELSE
     MACRO
         FIELD = 0,7 %;
%FI
```

Assuming COND_FLAG is a compile-time expression with a value of zero, the following source code would be generated:

```
MACRO
     FIELD = 0,7 %;
```

Code that is discarded as a result of a conditional compilation is never seen by the compiler and therefore may contain syntactic errors that will not be detected. The code may also contain partial declarations that will not be processed until substitution is complete. For example, it is legal to say:

```
%IF VISIBLE
%THEN
     GLOBAL
          X
%ELSE
     OWN
          X
%FI
          : VECTOR[10];
```

It is equally important to understand that compile-time functions and conditional compilations occurring in macro declarations are not evaluated and therefore do not take effect until the time of the macro call.

COMPILE-TIME DECLARATIONS

A compile-time declaration creates a symbolic name whose value can be changed at any time during the compilation of the program. In all other respects, a compile-time name is equivalent to a literal name and can be used in the same ways that a literal name can be used.

A compile-time name is declared with the keyword COMPILETIME. For example,

```
COMPILETIME
     FLAG = 0;
```

declares the name FLAG as a compile-time name with the value zero. As such, it is equivalent to a LITERAL declaration with the exception that its value may subsequently be changed.

The function %ASSIGN(compile-time-name, compile-time-constant) modifies the value of the compile-time name by assigning the specified compile-time constant as its value. The function %ASSIGN(...) is replaced by the null lexeme (i.e., it disappears) after the assignment is made; consequently, it should not be followed by a ";" since the semicolon would not disappear and might introduce syntactic complications. An example of this function is

```
%ASSIGN(FLAG,1)
```

which resets the compile-time literal FLAG to the value one, and

```
%ASSIGN(FLAG, FLAG + 2)
```

which resets the current value to the value of FLAG plus two. Note that FLAG is a compile-time literal and, therefore, the dot operator is not used to obtain its value. Together, these two functions provide added flexibility in controlling the compilation process. For example,

```
COMPILETIME
    COUNT = 0;
...

%ASSIGN(COUNT, COUNT + 1)
...

%IF COUNT GTR 4
  %THEN
        code segment one
        ...

%ELSE
        code segment two
        ...

%FI
```

allows the user to select a specific source code based upon a compile-time counter which is incremented during the course of the compilation.

LEXICAL FUNCTIONS

The conditional compilation mechanism is further enhanced by a number of special lexical functions. They include test functions, which are used to evaluate compile-time constants; advisory functions, which are used to generate compile-time messages; and macro functions, which are used in conjunction with the conditional compilation of macros.

TEST FUNCTIONS

Each test function contains one or more arguments which are evaluated to determine whether a given condition is true or false. The test function is then replaced by a one (true) or zero (false) as appropriate.

The function %NULL(argument,...) returns a one if all of its arguments are null or a zero if one or more of its arguments are not null. For example,

```
MACRO
    NR_LINES(LINE_SIZE)=
        %IF NOT %NULL(LINE_SIZE)
        %THEN
        ,   LINES = .CHARS / LINE_SIZE
        %FI  %;
```

generates a null if the macro NR_LINES is called without an argument because %NULL returns a one (i.e., NOT %NULL returns a zero).

The function %DECLARED(name) returns a one if its argument is a user declared name and a zero if it is not. For example, the test condition in

```
OWN
    STORE;
...
%IF %DECLARED(STORE)
%THEN
    TABLE[.K] = .STORE;
%FI
```

returns a one because STORE is a user-defined name and, consequently, the THEN sequence is included in the compilation. This function accepts only one argument.

The function %SWITCHES(argument,...) returns a one if the standard BLISS compilation switches included as arguments are true (activated) at that time, and a zero if one or more are not. For example,

```
%IF %SWITCHES(DEBUG)
%THEN
    INCR I FROM 0 TO 9 DO
        TTY_PUT_INTEGER(.TABLE[.I],10,10);
%FI
```

succeeds and generates the INCR loop if the DEBUG switch is activated for the compilation.

The function %IDENTICAL(arguments,...) requires a minimum of two arguments and returns a one if these arguments are exactly the same, or a zero if they are not. "Exactly", in this context, requires that both the value and the type of literal agree. This

is illustrated by the following example, where each of the test
conditions fail for the indicated reason:

```
LITERAL
    SIXFIVE = 65;

...

%IF  %IDENTICAL(65,%C'A')...              !NUMBER VS STRING
                                          ! LITERAL
%IF  %IDENTICAL(65,%O'101')...            !ONE VS TWO LEXEMES
%IF  %IDENTICAL(65,SIXFIVE)...            !LITERAL VS NAME
%IF  %IDENTICAL(%ASCII'A',%C'A')...       !TWO VS ONE LEXEME
```

The function %BLISS(argument) accepts the generic name of a
BLISS compiler, i.e.,

```
BLISS16
BLISS32
BLISS36
```

and returns a one if the compiler is processing the program or a
zero if another compiler is processing the program. For example,

```
%IF  %BLISS(BLISS32)
%THEN
    CHAR_PER_WORD = 4;
%ELSE
    %IF  %BLISS(BLISS36)
    %THEN
        CHAR_PER_WORD = 5;
    %ELSE
        CHAR_PER_WORD = 2;
    %FI
%FI
```

results in the appropriate source code being generated to assign
to CHAR_PER_WORD the correct number of characters per word,
depending upon the compiler which is processing the source code.

ADVISORY FUNCTIONS

The advisory functions provide a means to generate
compile-time diagnostic and/or other messages which will appear in
the program listing being generated. These functions do not
produce source code and therefore have no run-time effect on the
program being compiled.

The function %ERROR(quoted-string,...) produces an error diagnostic in the listing which includes the quoted string as its message. For example,

```
%IF NOT %BLISS(BLISS36)
%THEN
    %ERROR('YOU HAVE THE WRONG COMPILER');
```

generates a compile-time error and the diagnostic message 'you have the wrong compiler' if the BLISS-36 compiler is not the one being used. Because the error indicator is incremented when this function is invoked, no object module is generated.

The function %WARN(quoted-string,...) produces a warning diagnostic which includes the quoted string. The warning indicator is incremented and an object module is produced.

The function %INFORM(quoted-string,...) produces a diagnostic but does not increment the error or warning indicators. The quoted string is output on both the listing file and the terminal as a message.

The function %PRINT(quoted-string,...) prints the quoted string in the compilation listing but does not generate a diagnostic. For instance,

```
%IF %SWITCHES(DEBUG)
%THEN
    %PRINT('DEBUG SWITCH ON ');
%ELSE
    %INFORM('DEBUG SWITCH NOT ACTIVIATED')
%FI
```

prints a message in the compilation listing, reminding the user that the debug switch was activated, or else prints a diagnostic, indicating that the debug switch was not activated. The message generated by %INFORM is output to the terminal and included in the compilation listing; whereas the %PRINT message appears in the compilation listing only.

The last advisory function is %ERRORMACRO(quoted-string), which provides a method to stop macro processing within complex macros when there appears to be no reason to continue normal expansion. An error diagnostic is produced which includes the quoted string and the function terminates all currently active macro expansions. For example,

```
MACRO
    AVG_TWO(I,J)[]=
        %IF NOT (%LENGTH MOD 2)
        %THEN
            I = (.I * .J)/2;
            AVG_TWO(%REMAINING)
        %ELSE
            %ERRORMACRO('ODD NUMBER OF ARGUMENTS')
        %FI %;
```

terminates the macro expansion, if an odd number of arguments is
passed to the macro call.

MACRO FUNCTIONS

     The macro functions have no meaning outside of a macro
expansion;   however, within a macro expansion, these functions
terminate the expansion of the macro in which they are contained.

     The function %EXITITERATION has no arguments.   In   an
iterative macro expansion, this function terminates the expansion
of the current iteration and is replaced by a null.   In a
noniterative macro %EXITITERATION is functionally equivalent to
%EXITMACRO.

     The function %EXITMACRO also has no arguments. It terminates
the expansion of the macro in which it is contained and is
replaced with a null. Using this function, the example above
could have been written:

```
MACRO
    AVG_TWO(I,J)[]=
        %IF NOT (%LENGTH MOD 2)
        %THEN
            I = (.I * .J)/2;
            AVG_TWO(%REMAINING)
        %ELSE
            %INFORM('ODD NUMBER OF ARGUMENTS')
            %EXITMACRO
        %FI %;
```

The advantage of this construction is that an object module is
produced although the macro expansion was terminated; whereas in
the former example an object module would not have been produced
because the error indicator was incremented.

This page is for notes.

Exercises

1.   Write an initialization macro named INIT that initializes
the fields

```
BLISS-16 -
     Fields:  A = <0,8>
              B = <8,8>
BLISS-32 -
     Fields:  A = <0,8>
              B = <8,8>
              C = <16,8>
              D = <24,8>
BLISS-36 -
     Fields:  A = <0,12>
              B = <12,6>
              C = <18,10>
              D = <28,8>
```

depending upon the specific compiler used.   Print a warning
message if any of the required arguments are missing.

2.   Write an initialization macro named INIT that initializes
the fields in Exercise 1) depending upon the specific compiler
used. In addition to initializing the fields, store the values of
the arguments in a table named SAVE if the debug switch is
activiated.

3.   Same as Exercises 1) and 2) and, in addition, terminate
the macro with an error diagnostic if the user has not declared a
variable named SAVE.

This page intentionally left blank.

Solutions

```
1)    MACRO
          INIT(A,B,C,D)=
              %IF  %BLISS(BLISS16)
              %THEN
                  %IF  %NULL(A) OR  %NULL(B)
                  %THEN
                      %WARN('MISSING ARGUMENT')
                  %ELSE
                      ((B) ^ 8 OR (A))
                  %FI
              %ELSE
                  %IF  %BLISS(BLISS32)
                  %THEN
                      %IF  %NULL(A) OR  %NULL(B) OR  %NULL(C) OR
                           %NULL(D)
                      %THEN
                          %WARN('MISSING ARGUMENT')
                      %ELSE
                          ((D) ^ 24 OR (C) ^ 16 OR (B) ^ 8 OR (A))
                      %FI
                  %ELSE
                      %IF  %BLISS(BLISS36)
                      %THEN
                          %IF  %NULL(A) OR  %NULL(B) OR  %NULL(C)
                               OR  %NULL(D)
                          %THEN
                              %WARN('MISSING ARGUMENT')
                          %ELSE
                              ((D) ^ 28 OR (C) ^ 18 OR (B) ^ 12
                              OR (A))
                          %FI
                      %FI
                  %FI
              %FI  %;
```

```
2)      COMPILETIME
            COUNT = 0;
    ...

    MACRO
        INIT(A,B,C)=
            %IF %SWITCHES(DEBUG)
            %THEN
                SAVE[COUNT] = (A);
                %ASSIGN(COUNT, COUNT + 1)
                SAVE[COUNT] = (B);
                %ASSIGN(COUNT, COUNT + 1)
            %FI

            %IF %BLISS(BLISS16)
            %THEN
                (B ^ 8 OR A)
            %ELSE
                %IF %BLISS(BLISS32)
                %THEN
                    %IF SWITCHES(DEBUG)
                    %THEN
                        SAVE[COUNT] = (C);
                        %ASSIGN(COUNT, COUNT + 1)
                        SAVE[COUNT] = (D);
                        %ASSIGN(COUNT, COUNT + 1)
                    %FI
            ((D) ^ 24 OR (C) ^ 16 OR (B) ^ 8 OR (A))
                %ELSE
                    %IF %BLISS(BLISS36)
                    %THEN
                        %IF SWITCHES(DEBUG)
                        %THEN
                            SAVE[COUNT] = (C);
                            %ASSIGN(COUNT,COUNT + 1)
                            SAVE[COUNT] = (D);
                            %ASSIGN(COUNT,COUNT +1)
                        %FI
            ((D) ^ 28 OR (C) ^ 18 OR (B) ^ 12 OR (A))
                    %FI
                %FI
            %FI  %;
```

```
3)   COMPILETIME
         COUNT = 0;
     ...

     MACRO
         INIT(A,B,C)=
              %IF NOT %DECLARED(SAVE)
              %THEN
                  ERRORMACRO('SAVE NOT DECLARED')
              %FI

              %IF %SWITCHES(DEBUG)
              %THEN
                  SAVE[COUNT] = (A);
                  %ASSIGN(COUNT, COUNT + 1)
                  SAVE[COUNT] = (B);
                  %ASSIGN(COUNT, COUNT + 1)
              %FI

              %IF %BLISS(BLISS16)
              %THEN
                  ((B) ^ 8 OR (A))
              %ELSE
                  %IF %BLISS(BLISS32)
                  %THEN
                      %IF SWITCHES(DEBUG)
                      %THEN
                          SAVE[COUNT] = (C);
                          %ASSIGN(COUNT, COUNT + 1)
                          SAVE[COUNT] = (D);
                          %ASSIGN(COUNT, COUNT + 1)
                      %FI
                  ((D) ^ 24 OR (C) ^ 16 OR (B) ^ 8 OR (A))
                  %ELSE
                      %IF %BLISS(BLISS36)
                      %THEN
                          %IF SWITCHES(DEBUG)
                          %THEN
                              SAVE[COUNT] = (C);
                              %ASSIGN(COUNT, COUNT + 1)
                              SAVE[COUNT] = (D);
                              %ASSIGN(COUNT, COUNT + 1)
                          %FI
                      ((D) ^ 28 OR (C) ^ 18 OR (B) ^ 12 OR (A))
                      %FI
                  %FI
              %FI  %;
```

This page intentionally left blank.

Unit Test

Write a macro named INIT that uses conditional compilation to initialize a vector depending upon the compiler used, so that:

```
BLISS 36 -
       Fields:  A = <0,18>
                B = <18,18>
BLISS 32 -
       Fields:  A = <0,8>
                B = <8,8>
                C = <16,16>
```

In addition, the solution should terminate the compilation with an error diagnostic if:

* required arguments are missing
* the macro is called more than some arbitrary
   number of times (use two compile constants named
   LIMIT and COUNT)

Assume COUNT is initially set to zero; LIMIT is set to an arbitrary value at the beginning of each vector initialization; and COUNT is incremented by one on each call.

This page intentionally left blank.

OPTIMIZATION
MODULE II-8

Course Map
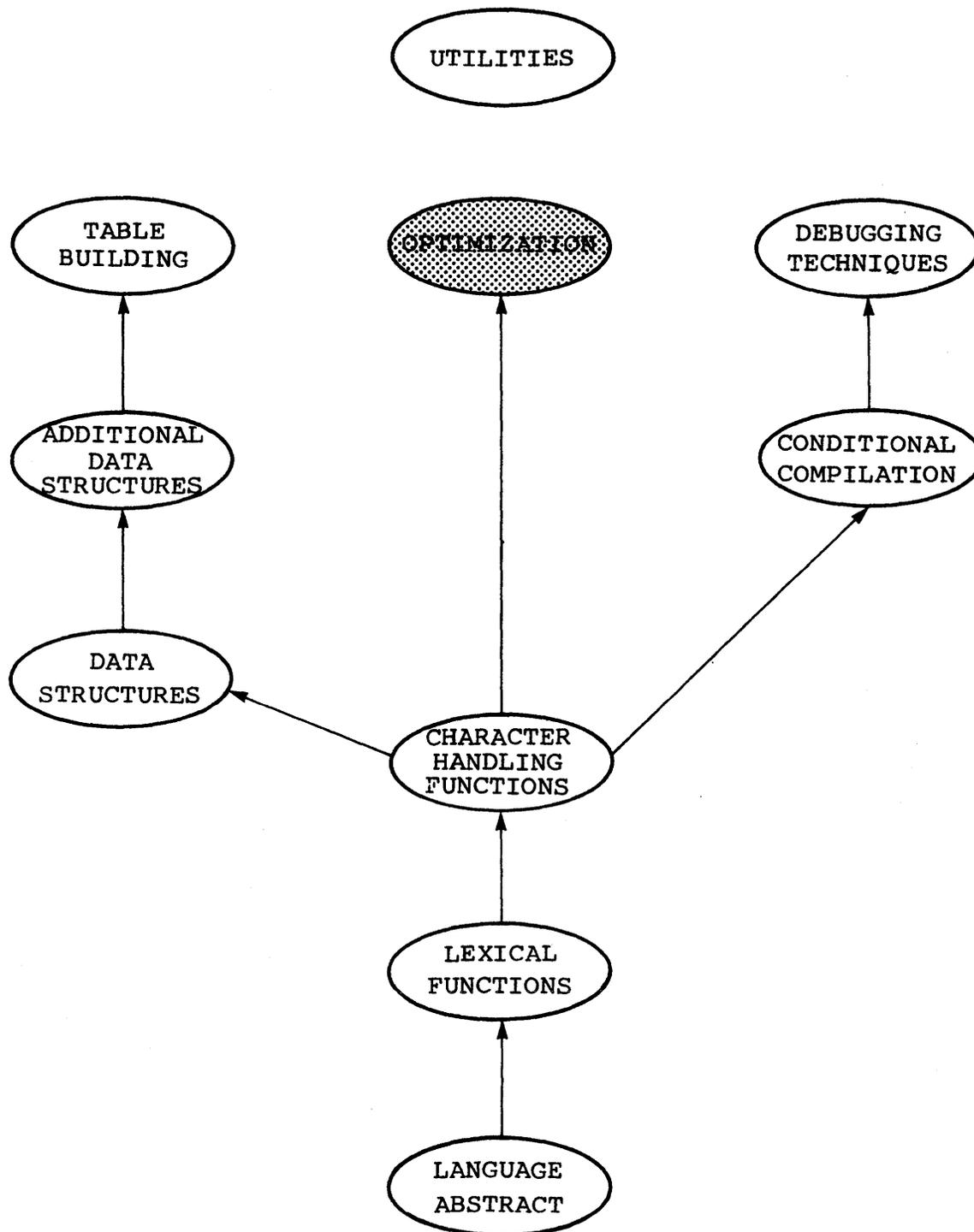
Introduction

This unit presents source-code optimizations intended for programmers that need to minimize space requirements or want to produce highly optimized code.

Objectives

Given an algorithm coded in BLISS-32, be able to rewrite the source code so that the object code produced by the compiler is more highly optimized than the original with respect to program size.

─────────────────────── **Sample Test Items** ───────────────────────

   Modify the following BLISS-32 source code to reduce the
current routine size of the object code produced by the compiler
by at least 11 bytes:

```
    MODULE TEST(MAIN=SP_CHAR)=
    BEGIN
    EXTERNAL ROUTINE
        PROC;

    EXTERNAL LITERAL
        NULL,                                       !=0
        SETONE;                                     !=1

    GLOBAL
        FLAG;

    ROUTINE SP_CHAR(TYPE)=                  !ROUTINE SIZE 86 BYTES
    !++
    !   THIS ROUTINE CONVERTS SELECTED SPECIAL CHARACTERS TO
    !   THEIR FUNCTIONAL EQUIVALENT AND RETURNS THE VALUE.
    !--
        BEGIN
        LOCAL
            TEMP;

        TEMP = .TYPE;

        SELECT .TYPE OF
            SET
            [%C'[']:        TEMP = %C'(';
            [%C'\']:        TEMP = %C'/';
            [%C']']:        TEMP = %C')';
            [OTHERWISE]:    FLAG = SETONE;
            [ALWAYS]:       (FLAG = NULL; PROC(.TEMP));
            TES;
        RETURN .TEMP;
        END;
    END
    ELUDOM
```

This page is for notes.

The BLISS compilers are optimizing compilers which in general produce highly=optimized object code. The code generated by the compilers compares favorably with that produced by competent system programmers using assembly language. As a consequence, BLISS programmers are encouraged to write source code so that it can be easily read by other competent programmers, and to leave the optimization to the compiler. That remains sound advice even though this unit discusses optimization! For most applications, coherent code is preferred even at the expense of program size and/or execution time.

There are, however, many different options available when coding any given algorithm, and these choices tend to be arbitrary. This unit therefore attempts to identify source code trade=offs or selective options that enable programmers, when necessary, to conserve space and/or time. In addition, this unit provides coding conventions which make it possible for the compiler to produce more efficient object code.

Note

You are cautioned that the assembly code depicted in this unit was derived using version 1D(114) of the BLISS=32 compiler and may not be duplicable with other BLISS compilers. This is due to continual improvements being made to the compilers and the on=going effort to produce more highly optimized code. The focus of each example is to show that a savings of generated code is possible; the actual number of bytes saved will vary, depending upon the compiler.

NOVALUE ROUTINES

The NOVALUE attribute was introduced in an earlier unit and has been used extensively throughout subsequent units in routines that did not return a value. Although this attribute is not mandatory, even when a routine does not return a value, its use is recommended since it can effect savings in both program size and execution time. When a routine is identified as NOVALUE, the compiler does not provide for return value generation and can therefore avoid allocating storage for potential return values. The compiler does not have to generate instructions that would otherwise be necessary to move and store these values. Of course, the actual savings for any given routine are highly dependent upon

the complexity of the routine. BLISS optimization occurs at the
routine level. Consider, as an example, the routine:

```
ROUTINE NVR(X,Y)=              NVR:
    BEGIN                            .WORD    ^M<>
    IF .Y EQL 1                      MOVL     8(AP),R0
    THEN                             CMPL     R0,#1
        .X = 5                       BNEQ     1$
    ELSE                             MOVL     #5,@4(AP)
        '.Y = 3                      MOVL     #5,R0
    END;                             RET
                             1$:     MOVL     #3,(R0)
                                     MOVL     #3,R0
                                     RET
```

Because this routine does not contain the NOVALUE attribute, it
requires code to move the potential values of X (i.e., five) and
of Y (i.e., three) to the return register R0. The routine size of
the resulting object code is 26 bytes. The same routine with the
NOVALUE attribute is six bytes smaller and no longer requires the
longword moves to register R0. In addition, someone reading the
program would know immediately that this routine does not return a
value.

    For routines that do return a value, the keyword RETURN
improves program readability and also provides additional
information to the compiler, which can facilitate optimization by
identifying the specific variable that will contain the value to
be returned. Being able to identify which variable will contain
the return value, prior to code generation, the compiler can often
assign it to the return register. Even when no saving results, no
additional code is generated! For example, the routine

```
ROUTINE RTN=                   RTN:
    BEGIN                            .WORD    ^M<>
    LOCAL                            MOVL     Y,R1
        X;                           BLBC     R1,1$
    IF .Y                            MOVL     R1,R0
    THEN                             RET
        X = .Y               1$:     MNEGL    R1,R0
    ELSE                             RET
        X = =.Y;
    .X
    END;
```

which does not explicitly identify the return value requires seven
instructions and has a program size of 18 bytes; whereas the same
routine, using an explicit RETURN expression which eliminates the

local X, requires two fewer instructions and is reduced in size by
four bytes:

```
ROUTINE RTN=                        RTN:
    BEGIN                                   .WORD    ^M<>
    RETURN (                                MOVL     Y,RØ
    IF .Y                                   BLBS     RØ,1$
    THEN                                    MNEGL    RØ,RØ
        .Y                          1$:     RET
    ELSE
        -.Y );
    END;
```

This code is also easier to read.

It should be noted that the examples in this unit are
necessarily small and that this limits many opportunities for
optimization which can occur as a result of using the suggested
coding techniques.

EXTERNAL LITERALS

As a general rule, external literals should not be used: it
is better to define the values of the literals locally than to
reference them through external literals.   If external literals
are essential, they should contain the size of each field.
Without size information, the compiler assumes a fullword literal.
The extensive use of fullword literals can produce a significant
increase in the size of the program.   This is especially
applicable if these literals have small numeric values. For
example, consider the declaration:

```
LITERAL
    TYPE_A = 1,
    TYPE_B = 2,
...
    TRUE = 1;
```

The use of fullword external literals in comparisons and in other
expressions effect both program space execution time.   For
example, using the external literals

```
EXTERNAL LITERAL
    ONE,
    TRUE;
```

the routine (BLISS-32)

```
    ROUTINE EXT=                        EXT:
        BEGIN                                   .WORD   ^M<>
        LOCAL                                   CLRL    RØ
            X;                                  CMPL    #TRUE,#1
        X = Ø;                                  BNEQ    1$
        IF TRUE EQL 1                           ADDL2   #ONE,RØ
        THEN                            1$:     CLRL    RØ
            X = .X + ONE;                       RET
        END;
```

has a size of 23 bytes. On the other hand, the same routine using
external literals that include the field size

```
    EXTERNAL LITERAL
        ONE:UNSIGNED(4),
        TRUE:UNSIGNED(4);
```

generates the same code but has a routine size of only 15 bytes.
Of   course,   completely   eliminating   these external literals
decreases the routine size to nine bytes and produces better code:

```
                            EXT:
                                    .WORD   ^M<>
                                    CLRL    RØ
                                    INCL    RØ
                                    CLRL    RØ
                                    RET
```

MINIMIZING TEMPORARIES

    In general, the use of temporary storage is also discouraged.
This will probably appear strange to more experienced higher-level
programmers, particularily those who are accustomed to FORTRAN
type languages. For example, code equivalent to the BLISS

```
ROUTINE TEX=                          TEX:
    BEGIN                                       .WORD      ^M<>
    LOCAL                                       ADDL3      B,A,RØ
        TEMP;                                   SUBL2      C,RØ
    TEMP = .A + .B - .C;                        CMPL       RØ,#100
    IF .TEMP GTR 100                            BLEQ       1$
    THEN                                        MOVL       #1,RØ
        1                                       RET
    ELSE                              1$:       CLRL       RØ
        Ø                                       RET
    END;
```

is often necessary. The variable TEMP is used in this example to
provide a single variable on which to perform the conditional
test. Assuming that TEMP is a local variable, the routine size is
31 bytes.   In this example the compiler is able to allocate TEMP
to a register, because this routine is small and, therefore, the
object code is identical to that produced for the following source
code without the temporary variable:

```
ROUTINE TEX=
BEGIN
IF (.A + .B - .C) GTR 100
THEN
    1
ELSE
    Ø
END;
```

However, as the complexity of the algorithm and the number of
temporary variables increases, the ability of the compiler to best
utilize the existing registers and, therefore, to produce optimum
code is severly hampered.   This is particularily true of
control-flow values since these values need not necessarily be
allocated storage. Control-flow expressions evaluate to a true or
false indicator which determines whether or not to jump.

     If static storage had been used in the original example
instead of local storage, the routine size would have increased
five bytes to 36 bytes. This is depicted below:

```
                                 TEX:
                                      .WORD      ^M<>
                                      ADDL3      B,A,RØ
                                      SUBL3      C,RØ,TEMP
                                      CMPL       TEMP,#100
                                      BLEQ       1$
                                      MOVL       #1,RØ
                                      RET
                                 1$:  CLRL       RØ
                                      RET
```

A more reasonable example that involves several temporary
variables and which does result in savings, even when using local
variables, is shown below:

```
ROUTINE CMP=                CMP:
    BEGIN                        .WORD      ^M<R2>
    LOCAL                        MULL3      #10,X,R1
        A,                       MCOML      Z,R2
        B,                       BICL3      R2,Y,RØ
        C;                       SUBL3      #20,Y,R2
    A = .X * 10;                 BISL2      R1,RØ
    B = .Z AND .Y;               BISL3      R2,RØ,R1
    C = .Y = 20;                 BLBC       R1,1$
    A = .A OR .B OR .C;          MOVL       #1,RØ
    IF .A                        RET
    THEN                    1$:  CLRL       RØ
        1                        RET
    ELSE
        Ø
    END;
```

The temporary variables in this example require five additional
bytes of storage and also preclude the compiler from generating
code that could avoid evaluation of unnecessary expressions. Note
the difference when the temporary variables are eliminated:

```
ROUTINE CEX=                CEX:
    BEGIN                        .WORD      ^M<>
    IF (.X * 10) OR              MULL3      #10,X,RØ
       (.Z AND .Y) OR            BLBS       RØ,2$
       (.Y = 20)                 BLBC       Z,1$
    THEN                         BLBS       Y,2$
        1                   1$:  SUBL3      #20,Y,RØ
    ELSE                         BLBC       RØ,3$
        Ø                   2$:  MOVL       #1,RØ
    END;                         RET
                            3$:  CLRL       RØ
                                 RET
```

In addition to significantly reducing the routine size, the code produced is better optimized. It now evaluates only the expressions necessary to satisfy the test condition.

There is, however, a trade-off between source code optimization and obscurity. After all, production programs must be maintained and programmers attempting to maintain code that has been unnecessarily "tweaked" may have difficulties understanding and, subsequently, modifying it. The Style Guide in the VAX-11 Software Engineering Manual should therefore be consulted for coding practices which are to be avoided.

Also applicable to a discussion of minimizing temporaries is the capability of the compiler to recognize common subexpressions, because the elimination of common subexpressions is a typical use of temporaries. The BLISS-32 compiler can and does recognize common subexpressions if they are written correctly. For example,

```
ROUTINE CSB=                    CSB:
    BEGIN                               .WORD   M<>
    IF .N                               MOVL    X,RØ
    THEN                                BLBC    N,1$
        BEGIN                           ADDL3   #1,RØ,X
        X = .X + 1;                     CLRL    Z
        Z = 0;                          BRB     2$
        END                     1$:     MOVL    #4,Z
    ELSE                                ADDL3   #1,RØ,X
        BEGIN                   2$:     CLRL    RØ
        Z = 4;                          RET
        X = 1 + .X;
        END
    END;
```

contains code which is common to both branches; however, as written, it was not identified by the compiler as a common subexpression. If the two expressions had been written identically each time, the compiler would have recognized them as common subexpressions and would have produced the following code:

```
ROUTINE CSB                        CSB:
        BEGIN                              .WORD    M<>
        IF .N                              ADDL3    #1,X,R0
        THEN                               MOVL     RO,X
            BEGIN                          BLBC     N,1$
            X = .X + 1;                    CLRL     Z
            Z = 0;                         BRB      2$
            END                    1$:     MOVL     #4,Z
        ELSE                       2$:     CLRL     R0
            BEGIN                          RET
            Z = 4;
            X = .X + 1;
            END
        END;
```

This routine is six bytes smaller than the previous version, which
required 38 bytes.

CROSS JUMPING

Common subexpressions are recognized as parts of expressions
which are identical, wherever they occur.   Another source of
optimization is cross jumping which occurs when common code can be
identified in two or more branches. For example, the routine GP
below

```
ROUTINE GP=                        GP:
        BEGIN                              .WORD    ^M<>
        IF .K GTR 0                        TSTL     K
        THEN                               BLEQ     1$
            (RTN(.X); K = .K + 1)          PUSHL    X
        ELSE                               CALLS    #1,RTN
            RTN(3)                         ADDL3    #1,K,R0
        END;                               MOVL     R0,K
                                           RET
                                   1$:     PUSHL    #3
                                           CALLS    #1,RTN
                                           RET
```

calls the function RTN in both the THEN and ELSE branches. As it
is written, it has a routine size of 37 bytes and requires 11
instructions. But by rearranging the routine call so that it is
the last entry in each branch, the compiler can combine the calls
to RTN. The resulting routine

```
ROUTINE GP=                        GP:
    BEGIN                               .WORD       ^M<>
    IF .K GTR 0                         MOVL        K,R0
    THEN                                BLEQ        1$
        (K = .K + 1; RTN(.X))           ADDL3       #1,R0,K
    ELSE                                PUSHL       X
        RTN(3)                          BRB         2$
    END;                        1$:     PUSHL       #3
                                2$:     CALLS       #1,RTN
                                        RET
```

is smaller in size by eight bytes and has only nine instructions.

Another example of cross jumping where several common
expressions are involved is depicted below:

```
ROUTINE CSX=                       CSX:
    BEGIN                               .WORD       ^M<>
    CASE N                              MOVAL       N,R0
        FROM 0 TO 2 OF                  CASEL       R0,#0,#2
        SET                     1$:     .WORD       2$=1$
        [0]:                            .WORD       3$=1$
            (X=.X+1;Y=.Y+1;Z=0);        .WORD       4$=1$
        [1]:                            MOVL        #3,Z
            (Y=1+.Y;Z=1;X=.X+1);        MOVL        #3,R0
        [2]:                            RET
            (Z=2;Y=.Y+1;X=1+.X);  2$:   INCL        X
        [OUTRANGE]:                     INCL        Y
            Z=3                         CLRL        Z
        TES                             CLRL        R0
    END;                                RET
                                3$:     INCL        Y
                                        MOVL        #1,Z
                                        BRB         5$
                                4$:     MOVL        #2,Z
                                        INCL        Y
                                5$:     ADDL3       #1,X,R0
                                        MOVL        R0,X
                                        RET
```

Notice that much of the code is duplicated for in=range values;
however, as it is written, this code was not identified by the
compiler as common and was therefore not eliminated. If the
common code had been arranged in a group and positioned at the end
of each block, as shown below,

```
ROUTINE CSX=                        CSX:
    BEGIN                               .WORD      ^M<>
    CASE N                              MOVAL      N,R0
       FROM 0 TO 2 OF                   CASEL      R0,#0,#2
       SET                      1$:     .WORD      2$=1$
       [0]:                             .WORD      3$=1$
          (Z=0;X=.X+1;Y=.Y+1);         .WORD      4$=1$
       [1]:                             MOVL       #3,Z
          (Z=1;X=.X+1;Y=.Y+1);         MOVL       #3,R0
       [2]:                             RET
          (Z=2;X=.X+1;Y=.Y+1);  2$:     CLRL       Z
       [OUTRANGE]:                      BRB        5$
          Z=3;                  3$:     MOVL       #1,Z
    TES;                                BRB        5$
    END;                       4$:     MOVL       #2,Z
                               5$:     INCL       X
                                        ADDL3      #1,Y,R0
                                        MOVL       R0,Y
                                        RET
```

the compiler could recognize this duplication and would generate
only one such sequence. The resulting routine is 13 bytes smaller
than the previous version.

SEQUENTIAL PROCESSING

    Another source of potential optimization occurs when
accessing   storage.   In   general,   the   compiler   can   optimize
accessing and/or storing operations if the code is sequentially
executed.   This means that the programmer should attempt to
perform like operations together.   For example,

```
ROUTINE SBX =                       SBX:
    BEGIN                               .WORD      M<>
    BV<0,4> = 0;                        BICB2      #15,BV
    T = .Z;                             MOVL       Z,T
    BV<7,1> = 0;                        BICB2      #224,BV
    BV<5,3> = 0;                        CLRL       R0
    END;                                RET
```

resets the three fields of the longword, BV but the code does not
perform the operation sequentially.   It is interrupted with an
unrelated assignment.   Because the impact of. this unrelated
assignment   can   not   be   anticipated by the compiler, the
interruption results in the execution of the first assignment. As
a result, the routine requires two instructions to store this data

into the three fields. Note that the two subsequent assignments
were consolidated into one instruction. The routine requires 23
bytes of storage.

    If, on the other hand, these assignments had been written
sequentially, as below,

```
    ROUTINE SBX                   SBX:
        BEGIN                         .WORD    M<>
        BV<0,4> = 0;                  BICB2    #239,BV
        BV<7,1> = 0;                  MOVL     Z,T
        BV<5,3> = 0;                  CLRL     R0
        T = .Z;                       RET
        END;
```

the compiler would be able to consolidate the three storage
operations into one instruction. The resulting code requires less
space by five bytes and is also faster.

CONDITIONALS

    In BLISS the user has many ways of coding a conditional
expression, including case, select, selectone, and nested
IF=THEN=ELSE. Besides the differences in readability, there are
also differences with respect to optimization. To demonstrate
this, consider the problem of counting the occurrences of the
numbers one through four stored in a vector named TABLE. Although
the solution is obviously well=suited to a case expression, it
could be accomplished with any one of the various conditional
expressions and therefore provides a reasonable comparision.

    As a case expression the routine

```
ROUTINE COND=                        COND:
    BEGIN                                    .WORD    ^M<R2>
    INCR I FROM 0 TO 999 DO                  CLRL     R2
        CASE .TABLE[.I]          1$:         CASEL    T[R2],#1,#3
            FROM 1 TO 4 OF       2$:         .WORD    3$-2$
            SET                              .WORD    4$-2$
            [1]:                             .WORD    5$-2$
                FRESHMAN();                  .WORD    6$-2$
            [2]:                             CALLS    #0,ERROR
                SOPHMORE();                  BRB      7$
            [3]:                 3$:         CALLS    #0,FRESHMAN
                JUNIOR();                    BRB      7$
            [4]:                 4$:         CALLS    #0,SOPHMORE
                SENIOR();                    BRB      7$
            [OUTRANGE]:          5$:         CALLS    #0,JUNIOR
                ERROR()                      BRB      7$
        TES                      6$:         CALLS    #0,SENIOR
    END;                         7$:         AOBLEQ   #999,R2,1$
                                             MNEGL    #1,R0
                                             RET
```

generates particularly fast code because it has an equivalent
machine instruction. Having a routine size of 64 bytes, it is
also the smallest possible configuration produced by any of the
conditional expressions below.

The equivalent select expression

```
ROUTINE COND=                          COND:
    BEGIN                                  .WORD    ^M<R2,R3,R4>
    INCR I FROM 0 TO 999 DO                CLRL     R2
        SELECT .I OF               1$:     MOVL     T[R2],R4
            SET                            CMPL     R4,#1
            [1]:                           BNEQ     2$
                FRESHMAN();                CLRL     R3
            [2]:                           CALLS    #0,FRESHMAN
                SOPHMORE();        2$:     CMPL     R4,#2
            [3]:                           BNEQ     3$
                JUNIOR();                  CLRL     R3
            [4]:                           CALLS    #0,SOPHMORE
                SENIOR();         3$:     CMPL     R4,#3
            [OTHERWISE]:                   BNEQ     4$
                ERROR()                    CLRL     R3
            TES                            CALLS    #0,JUNIOR
    END;                          4$:     CMPL     R4,#4
                                           BNEQ     5$
                                           CLRL     R3
                                           CALLS    #0,SENIOR
                                   5$:     BLBC     R3,6$
                                           CALLS    #0,ERROR
                                   6$:     AOBLEQ   #999,R2,1$
                                           MNEGL    #1,R0
                                           RET
```

requires 81 bytes and utilizes slower compare and branch
instructions. Although a select expression is probably easier to
read, the code produced strongly resembles the code generated for
the equivalent nested IF-THEN-ELSE construction:

```
ROUTINE COND=                        COND:
    BEGIN                                .WORD      ^M<R2,R3>
    INCR I FROM 0 TO 999 DO              CLRL       R3
        IF .I EQL 1              1$:     MOVL       T[R3],R2
        THEN                             CMPL       R2,#1
            FRESHMAN()                   BNEQ       2$
        ELSE                             CALLS      #0,FRESHMAN
            IF .I EQL 2                  BRB        6$
            THEN                 2$:     CMPL       R2,#2
                SOPHMORE()               BNEQ       3$
            ELSE                         CALLS      #0,SOPHMORE
            IF .I EQL 3                  BRB        6$
            THEN                 3$:     CMPL       R2,#3
                JUNIOR()                 BNEQ       4$
            ELSE                         CALLS      #0,JUNIOR
                IF .I EQL 4              BRB        6$
                THEN             4$:     CMPL       R2,#4
                    SENIOR()             BNEQ       5$
                ELSE                     CALLS      #0,SENIOR
                    ERROR()              BRB        6$
    END;                         5$:     CALLS      #0,ERROR
                                 6$:     AOBLEQ     #999,R3,1$
                                         MNEGL      #1,R0
                                         RET
```

However, the routine with the nested IF=THEN=ELSE expression
requires only 75 bytes and has the advantage of exiting the
routine after executing the expressions following the match.   The
IF=THEN=ELSE expression generates identical code to that produced
for the equivalent selectone expression:

```
        ROUTINE COND=
            BEGIN
            INCR I FROM 0 TO 999 DO
                SELECTONE .TABLE[.I] OF
                    SET
                    [1]:
                        FRESHMAN();
                    [2]:
                        SOPHMORE();
                    [3]:
                        JUNIOR();
                    [4]:
                        SENIOR();
                    [OTHERWISE]:
                        ERROR();
                    TES
            END;
```

Exercises

In each of the exercises below, modify the BLISS-32 source
code so that the routine size of each object module produced by
the compiler is reduced by at least six bytes:

```
1)   ROUTINE SX=                              !ROUTINE SIZE 67 BYTES
     !++
     !    THIS ROUTINE CHANGES SQUARE AND FANCY BRACKETS INTO
     !    THEIR EQUIVALENT PARENTHESIS.  "TYPE" IS STATIC STORAGE
     !    DECLARED IN AN OUTER BLOCK.
     !--
     BEGIN
     TYPE = (
         SELECT .TYPE OF
         SET
         [%C'[', %O'173']: %C'(';
         [%C']', %O'175%C']: %C')';
         [OTHERWISE]: 0
         TES)
       END;

2)   ROUTINE SP_CHAR(TYPE)=                   !ROUTINE SIZE 71
BYTES
     !++
     !    THIS ROUTINE CONVERTS SELECTED SPECIAL CHARACTERS TO
     !    THEIR FUNCTIONAL EQUIVALENTS AND RETURNS THAT VALUE.
     !--
     BEGIN
     EXTERNAL
         FLAG;

     EXTERNAL LITERAL
         SETONE;                                  !SETONE = 1

     LOCAL
         TEMP;

     SELECT .TYPE OF
         SET
         [%C'[']:              TEMP = %C'(';
         [%C'']:              TEMP = %C'';
         [%C']']:             TEMP = %C')';
         [OTHERWISE]:         BEGIN
                              FLAG = SETONE;
                              TEMP = .TYPE
                              END

         TES;
       .TEMP
     END;
```

This page intentionally left blank.

Solutions

1)  ```
    ROUTINE SX: NOVALUE=                    !ROUTINE SIZE 61 BYTES
    !++
    !    THIS ROUTINE CHANGES SQUARE AND FANCY BRACKETS INTO
    !    THEIR EQUIVALENT PARENTHESES.  "TYPE" IS STATIC STORAGE
    !    DECLARED IN AN OUTER BLOCK.
    !--
        BEGIN
        TYPE = (
            SELECTONE .TYPE OF
            SET
            [%C'[',%O'173']: %C'(';
            [%C']',%O'175']: %C')';
            [OTHERWISE]: 0
            TES)
        END;
    ```

2)  ```
    ROUTINE SP_CHAR(TYPE)=                   !ROUTINE SIZE 39 BYTES
    !++
    !    THIS ROUTINE CONVERTS SELECTED SPECIAL CHARACTERS TO
    !    THEIR FUNCTIONAL EQUIVALENTS AND RETURNS THAT VALUE.
    !--
        BEGIN
        EXTERNAL
            FLAG;

        EXTERNAL LITERAL
            SETONE: UNSIGNED(1);

        RETURN (
                CASE .TYPE
                    FROM %O'133' TO %O'135' OF
                    SET
                    [%C'\']: %C'/';
                    [%C'[']: %C'(';
                    [%C']']: %C')';
                    [OUTRANGE]: (FLAG = SETONE; .TYPE)
                    TES)
        END;
    ```

This page intentionally left blank.

Unit Test

Modify the following BLISS-32 source code so that the current
routine size of the object code produced by the compiler is
reduced by at least 11 bytes:

```
MODULE TEST(MAIN=
BEGIN
EXTERNAL ROUTINE
     PROC;

EXTERNAL LITERAL
     NULL,                               !=0
     SETONE;                             !=1

GLOBAL
     FLAG;

ROUTINE SP_CHAR(TYPE)=                   !ROUTINE SIZE 86 BYTES
!++
!    THIS ROUTINE CONVERTS SELECTED SPECIAL CHARACTERS TO
!    THEIR EQUIVALENT AND RETURNS THAT VALUE.
!--
     BEGIN
     LOCAL
         TEMP;

     TEMP = .TYPE;

     SELECT .TYPE OF
         SET
         [%C'[']:
             TEMP = %C'(';
         [%C'\']:
             TEMP = %C'/';
         [%C']']:
             TEMP = %C')';
         [OTHERWISE]:
             FLAG = SETONE;
         [ALWAYS]:
             (FLAG = NULL; PROC(.TEMP)
         TES;
     .TEMP
     END;
END
ELUDOM
```

This page intentionally left blank.

DEBUGGING TECHNIQUES

MODULE II-9

Course Map
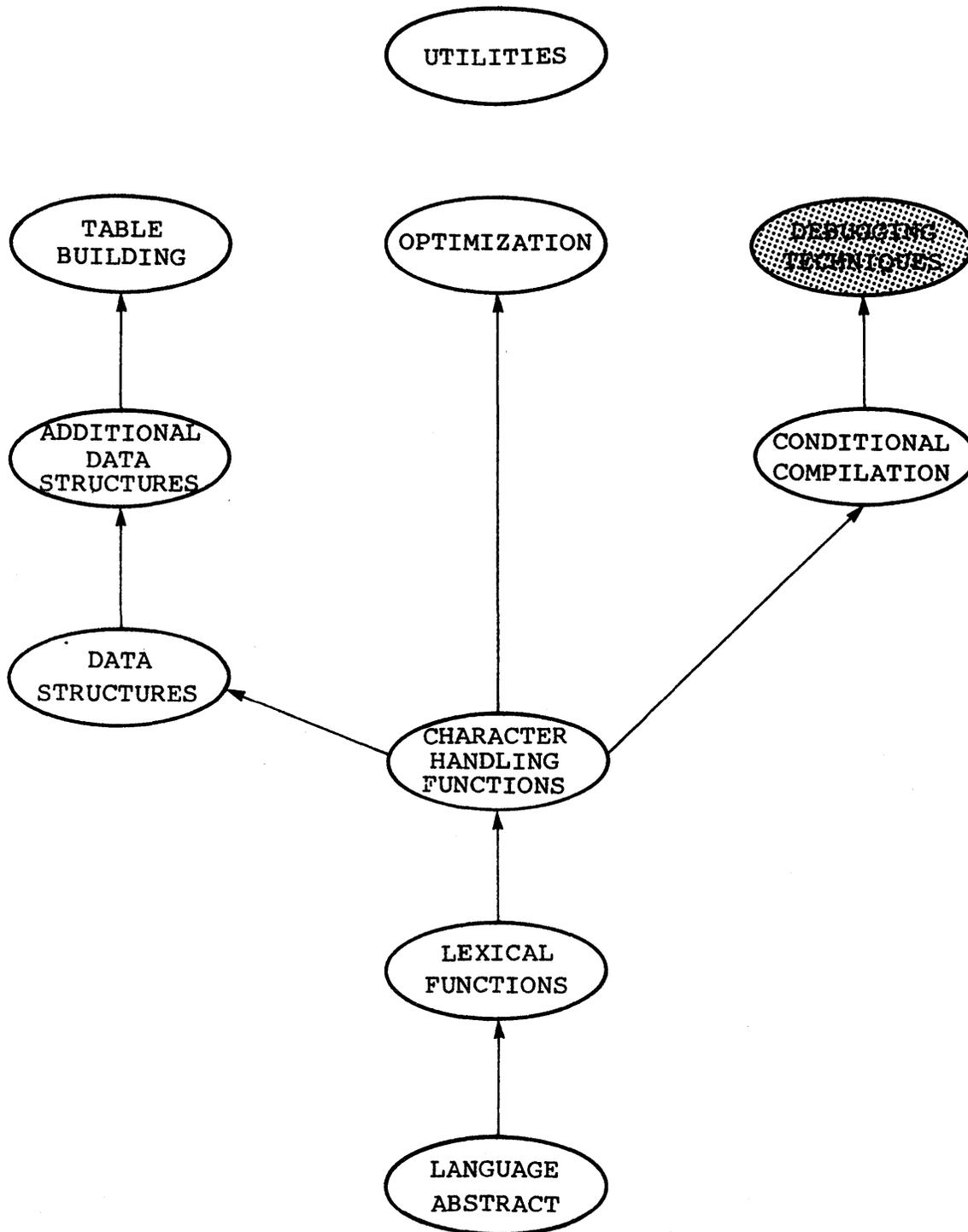
Introduction

    This unit illustrates a variety of code which can be added to
a module to facilitate subsequent testing and debugging. In
general, the code consists of macros constructed with the
conditional compilation mechanism.

─────────────────────── **Objectives** ───────────────────────

Given a module with one or more routines, be able to insert conditional code that can be used to monitor and debug the module.

─────────────────────── **Sample Test Items** ───────────────────────

Given the module TEST.BLI (on your disk area and in Attachment #1), which includes trace and monitor macros, insert the necessary macro calls to test and debug the program. Execute the program and identify the existing bugs.

─────────────────────── **Additional Resources** ───────────────────────

None

As most production programmers are aware, debugging can and usually does consume a significant part of the time required to produce a substantially error free program, i.e., a program containing no branches that:

* produce exception conditions that prematurely terminate execution
* result in an infinite loop
* produce an erroneous result

When one or more of these error conditions are encountered, the programmer has the problem of identifying and isolating the "bug". A variety of methods can be used including (but not limited to):

* reviewing the source code
* consulting with the local "expert"
* inserting code to output periodic messages/data
* using a debugger program

A point to consider when reviewing segments of source code, which logically (to you) should contain the problem but which appear correct, is that this code may in fact be correct. Therefore, after a thorough analysis of the segment has been conducted without finding the "bug", consider other reasons which could produce the error condition. This will frequently reveal the problem and save many frustrating hours.

A more aggressive approach in the attempt to produce error-free code is to program defensively. This includes:

* checking input parameters for validity
* making internal consistency checks
* providing built-in testing facilities

This unit concentrates on techniques for the latter and explores a variety of possible source code insertions which can be used at execution time to identify and isolate "bugs". This code is constructed using conditional compilation so as not to adversely affect the execution time of the finished program.

TRACE FACILITY

The mere isolation of a "bug" can be a significant task.
This is apparent to anyone who has had a program-stop with the
diagnostic:

?ILL MEM REF AT USER PC xxxxxx

Because this address may or may not be in a routine, the
diagnostic does not necessarily isolate the problem to a routine,
and, even if it did, it can not identify the recursion level at
which the error occurred. As a consequence, many hours can be
spent attempting to isolate the problem. The time is, of course,
dependent upon the experience of the programmer, the debugging
facilities available, and the size of the program. Infinite loops
can cause similar difficulties.

One solution is a simple trace mechanism to identify the
routine and the level of nesting at which the problem appears.
For example,

```
MACRO
    $TRACE(TIND, TNAME) =
        %IF  %SWITCHES(DEBUG)
        %THEN
            TTY_PUT_QUO(%STRING((TIND),' ROUTINE ',(TNAME),
                        %CHAR(%O'15',%O'12'))))
        %FI  %;
```

can be used with any routine such as

```
ROUTINE EVAL_STACK(STACK_ADDR,STACK_LEN,CURRENTOP) =
    BEGIN
    ...

    $TRACE('ENTER', 'EVAL_STACK');
    ...

    $TRACE('EXIT','EVAL_STACK');
    ...

    END;
```

to produce a trace of each call of that routine and each
subsequent exit by providing the messages:

```
ENTER ROUTINE EVAL_STACK
EXIT ROUTINE EVAL_STACK
```

Because the macro is constructed with a conditional compilation
dependent upon the debug switch, the code for the trace is
generated only when the debug switch is activated. This simple
mechanism can isolate a problem regardless of the nesting level at
which it appears. It therefore facilitates testing because the
routine order is generally known.

FORMATTED VARIABLES

Another potentially annoying problem, even using a debugger
program, is the display of data. Data frequently consists of
varying types (i.e., integer, ASCII, etc.) or is packed into
irregular size fields. For example, the block (32-bit machine)

```
┌─────────────────────────────────────────┐
│                NAME(32)                  │   TABLE:
├──────────┬──────────────────────────────┤
│ TYPE(2)  │          DATA(30)            │
└──────────┴──────────────────────────────┘
```

where

        NAME = 'R1'
        TYPE = 1 (designating an integer)
        DATA = 38 (decimal)

would, if dumped in hexadecimal with a conventional debugger, be
displayed as

        TABLE/   00005231
        TABLE+4/   40000026

or if dumped in ASCII mode:
        TABLE/   R1
        TABLE+4/   @ *!

Neither is extremely useful, and a long table displayed in either
format would be extremely frustrating. An alternative is to
insert code which interpretes the structure and displays the data
in the appropriate mode. For example (BLISS-32),

```
MACRO
    $DUMP_BLOCK =
        %IF %SWITCHES (DEBUG)
        %THEN
            INCR I FROM 0 TO 24 BY 8 DO
                IF .TABLE[0]<.I,8> EQL 0
                THEN
                    EXITLOOP
                ELSE
                    TTY_PUT_CHAR(.TABLE[0]<.I,8>);

            TTY_PUT_CRLF();

            IF TTY_PUT_INTEGER(.TABLE[1]<30,2>,10,1) EQL 1
            THEN
                TTY_PUT_INTEGER(.TABLE[1]<0,30>,10,10)
            ELSE
                BEGIN
                TTY_PUT_INTEGER(.TABLE[1]<15,15>,10,5);
                TTY_PUT_QUO(' / ');
                TTY_PUT_INTEGER(.TABLE[1]<0,15>,10,5);
                END
        %FI  %;
    . . .

ROUTINE EVAL_STACK(STACK_ADDR,STACK_LEN,CURRENT_OP)=
    BEGIN
    . . .
    $DUMP_BLOCK;
    . . .
    $DUMP_BLOCK;
    . . .
    END;
```

dumps the data in TABLE at the beginning and end of each call to
the routine EVAL_STACK in the format:

```
    R1
1           38
```

or if

```
    TYPE = 0   (designating a fraction)
    DATA = 300010 (in octal)
```

in the format:

```
    R1
0       3 / 8
```

This macro could also have been written with a few bells and
whistles:

```
    MACRO
        $DUMP_BLOCK =
            %IF %SWITCHES (DEBUG)
            %THEN
                TTY_PUT_QUO(%STRING(
                        'TYPE "Y" FOR DUMP',
                        %CHAR(%O'15,%O'12'))) ;

                IF TTY_GET_CHAR() EQL %C'Y'
                THEN
                    BEGIN
                    TTY_PUT_QUO('NAME:');

                    INCR I FROM 0 TO 24 BY 8 DO
                        IF .TABLE[0]<.I,8> EQL 0
                        THEN
                            EXITLOOP
                        ELSE
                            TTY_PUT_CHAR(.TABLE[0]<.I,8>);

                    TTY_PUT_CRLF();

                    IF .TABLE[1]<30,2> EQL 1
                    THEN
                        BEGIN
                        TTY_PUT_QUO('INTEGER: ');
                        TTY_PUT_INTEGER(.TABLE[1]<0,30>,10,10);
                        END
                    ELSE
                        BEGIN
                        TTY_PUT_QUO('FRACTION: ');
                        TTY_PUT_INTEGER(.TABLE[1]<15,15>,10,5);
                        TTY_PUT_QUO(' / ');
                        TTY_PUT_INTEGER(.TABLE[1]<0,15>,10,5);
                        END;
                    END
            %FI  %;
```

This version permits the user to determine at the time
"$DUMP_BLOCK" is invoked whether or not to dump the data in TABLE
by responding to the message:

    TYPE "Y" FOR DUMP

If a Y is typed, the following output occurs:

        NAME: R1
        INTEGER: 38

or

        NAME: R1
        FRACTION;   3 /   8

        It is important to note that the the macro is constructed
with a conditional compilation and, therefore:

    1) No debugging code is generated unless the appropriate
        switch is activitated; consequently, execution time is
        not affected when that switch is off.
    2) Using a macro permits a simple one line entry that makes
        the code convenient to generate and also permits it to
        be used for more than one routine.

MONITORING VALUES

        Another convenient facility is the ability to examine the
value of predetermined variables while the program is executing.
Minimally, one would like to know the initial and final values of
selected variables within each routine. A macro to accomplish
this is depicted below:

```
COMPILETIME
    TESTING = 1;
...

MACRO
    $LOOK(NAME, VAL)[]=
        %IF TESTING
        %THEN
            BEGIN
            TTY_PUT_QUO(%STRING((NAME), ':');
            TTY_PUT_INTEGER((VAL),10,10);
            TTY_PUT_CRLF();
            $LOOK(%REMAINING);
            END
        %FI  %;

...
```

```
        ROUTINE EVAL_STACK(STACK_ADDR, STACK_LEN, CURRENT_OP)=
            BEGIN
            ...

            $TRACE('ENTER', 'EVAL_STACK');
            $LOOK('STACK_ADDR', .STACK_ADDR,
                  'STACK_LEN', .STACK_LEN,
                  'CURRENT_OP', .CURRENT_OP);
    ...

            $TRACE('EXIT', 'EVAL_STACK');
            $LOOK('RETURN', .TEMP);
        RETURN .TEMP;
        END;
```

As used in the routine "EVAL_STACK" above,

```
    ENTER ROUTINE EVAL_STACK
    STACK_ADDR          xxx
    STCK_LEN            xxx
    CURRENT_OP          xxx
```

is output to the TTY each time the routine is entered and

```
    EXIT ROUTINE EVAL_STACK
    RETURN              xxx
```

is output as the final instruction prior to returning to the calling routine. Note that once the macro is constructed, the additional coding to insert this facility or any other is negligible especially considering the potential benefits.

ERROR CHECKING

A more subtle form of debugging occurs in bounds checking. This facility can be performed without obvious interaction, at least until an error is detected. Debugging in this manner is feasible since many parameters are predictable within reasonable limits and can therefore be bounded. For example,

```
    SELECT .OPNAMES OF
        SET
        [%C'(']: 0;
        ...

        [%C'=']: 9;
        [OTHERWISE]: ERROR(3);
```

produces an error message (e.g., "?ILL OP TYPE - RETYPE LINE") if
the contents of OPNAMES is not one of the acceptable characters;
however, the actual problem producing this message would have to
be further researched. However, if the code had been written

```
[OTHERWISE]  :  BEGIN
                ERROR(3);
                $LOOK('OPNAMES', .OPNAMES);
                END;
```

the output would have been displayed as

```
?ILL OP TYPE - RETYPE LINE
OPNAMES              xxx
```

depicting at least the contents of OPNAMES at the time of the
error and possibly indicating the source of the problem. Another
example of bounds checking performed only during debugging is
shown below:

```
MACRO
     $CHECK(NAME, NR, LOW, HIGH)=
         %IF %SWITCH(DEBUG)
         %THEN
              IF (NR) LSS (LOW) OR (NR) GTR (HIGH)
              THEN
                   BEGIN
                   TTY_PUT_QUO(%STRING(
                                     '%WARN        ',
                                     (NAME),
                                     '  EXCEEDS BOUNDS ...');
                   TTY_PUT_INTEGER((NR),10,10);
                   END
         %FI   %;
     ...
     $CHECK('VAL', .VAL, 0, 1000);
     IF .VAL LSS .CONDITION
     THEN
         ...
     ELSE IF .VAL GTR .PROCESS
          THEN
              ...;
```

Thus, as in the above example, critical values can be monitored
and reported, via the TTY, when they exceed reasonable limits
established by the programmer:

```
%WARN VAL EXCEEDS BOUNDS ...        xxxx
```

This is particularly beneficial when values are assumed never
to be outside of a given range, and, therefore, an error indicates
an oversight in design or a "bug" in the program.

Although many of the above mechanisms are possible with a
debugger, some "bugs" can elude detection without extensive
testing but can be detected from a display of intermediate data
and/or traces during execution. Most important, this code is
easily eliminated with a switch, yet it can be reactiviated for
subsequent testing or modification if necessary.

Note

A word of caution: the use of debugging code
which is not thoroughly tested can generate
"bugs" which are more elusive than those the
code was intended to detect. Therefore, use
only those features that contribute to
identifying potential problems, aid in
subsequent elimination of "bugs", and/or which
have been tested prior to use - i.e., keep it
simple. This can be accomplished by confining
the code to macros (or routines) and by
possibly maintaining it in tested REQUIRE
files.

This page is for notes.

ATTACHMENT #1

```
MODULE TEST (MAIN=R1) =
BEGIN

REQUIRE 'TUTIO';

COMPILETIME
    TESTING = 1;

LITERAL
    CR =            13,              ! CARRIAGE RETURN
    LF =            1Ø,              ! LINE FEED
    ASCII_ZERO = 48,                ! = "Ø"
    YES =           78,              ! = "Y"
    NO =            67;              ! = "N"

OWN
    X: INITIAL(Ø);

MACRO
    STRIP_CRLF() =
        TTY_GET_CHAR()   %;

MACRO
    $TRACE(TIND, TNAME) =
        %IF  TESTING
        %THEN
            TTY_PUT_QUO(%STRING((TIND),' ROUTINE ',(TNAME),
                                        %CHAR(CR,LF)))
        %FI  %;

MACRO
    $LOOK(NAME, VAL)[]=
        %IF  TESTING
        %THEN
            TTY_PUT_QUO(%STRING((NAME), ':'));
            TTY_PUT_INTEGER((VAL),1Ø,1Ø);
            TTY_PUT_CRLF();
            $LOOK(%REMAINING)
        %FI  %;

FORWARD ROUTINE
    R1,
    R2,
    R3,
    R4;
```

```
ROUTINE R1 =
    BEGIN
    LOCAL
        ANS,
        TEMP;

    IF .X NEQ 0
    THEN
        R3(0);

    TTY_PUT_QUO('TYPE A NUMBER 1 THRU 7  ');
    TEMP = TTY_GET_CHAR() - ASCII_ZERO;
    STRIP_CRLF;
    X = .TEMP;
    TTY_PUT_CRLF();
    TTY_PUT_QUO('YOU TYPED ...  ');
    TTY_PUT_CHAR(.TEMP+ASCII_ZERO);
    TTY_PUT_CRLF();
    TTY_PUT_QUO('TYPE "Y" FOR YES OR "N" FOR NO  ');
    ANS = TTY_GET_CHAR();
    TTY_PUT_CRLF();
    STRIP_CRLF;

    IF .ANS EQL YES
    THEN
        R2(.TEMP)
    ELSE
        IF .ANS EQL NO
        THEN
            R3(.TEMP-1);

    IF .X NEQ 0
    THEN
        R1();

    R4();
    END;
```

```
    ROUTINE R2(ARH)=
        BEGIN
        IF .ARH NEQ .X
        THEN
            BEGIN
            TTY_PUT_QUO('YOU CHEATED');
            TTY_PUT_CRLF();
            R1();
            END;

        TTY_PUT_QUO('MODULUS 4 OF YOUR INPUT IS ??  ');

        IF (TTY_GET_CHAR() - ASCII_ZERO) NEQ R3(ARH)
        THEN
            BEGIN
            STRIP_CRLF;
            R1();
            END;

        STRIP_CRLF;
        END;


    ROUTINE R3(A)=
        BEGIN
        IF .A EQL .X
        THEN
            X = 0
        ELSE
            R2(.A);

        RETURN (.A MOD 2);
        END;


    ROUTINE R4 =
        BEGIN
        TTY_PUT_QUO('EXIT');
        TTY_PUT_CRLF();
        END;
    END
    ELUDOM
```

This page is for notes.

Exercises

1. Write two macros $SAVE and $CHANGE which can be used
conditionally with the debug switch to monitor a single variable.
The macro $SAVE should save the initial value of the variable at
the time of the macro call and the macro $CHANGE should report its
old and current value if a change has occurred.

2. Write the macro $PAUSE which would allow a user to
conditionally invoke the debugging routine DUMP_STRU which dumps
global data structures. The macro should be conditional on the
COMPILETIME constant TESTING.

This page intentionally left blank.

Solutions

```
1)   MACRO
        $SAVE (VAL) =
            %IF %SWITCHES (DEBUG)
            %THEN
                LOCAL
                    %NAME(%STRING ('S$', (VAL)));
                %NAME(%STRING ('S$', (VAL)) = .(VAL)
            %FI  %,

        $CHANGE (VAL) =
            %IF %SWITCHES (DEBUG)
            %THEN
                IF .%NAME(%STRING ('S$', (VAL)) NEQ .(VAL)
                THEN
                    BEGIN
                    TTY_PUT_QUO(%STRING ((VAL), 'OLD');
                    TTY_PUT_INTEGER(.%NAME(%STRING ('S$',
                                        (VAL)),10,5);
                    TTY_PUT_QUO('NEW');
                    TTY_PUT_INTEGER(.(VAL),10,5);
                    END;
            %FI  %;


2)   MACRO
        $PAUSE =
            %IF TESTING
            %THEN
                TTY_PUT_QUO('TYPE "Y" TO CALL DEBUG ROUTINE ');
                IF TTY_GET_CHAR() EQL %C'Y'
                THEN
                    DUMP_STRU();
                STRIP_CRLF();
            %FI  %;
```

Note: The routine (or macro) STRIP_CRLF strips the carriage
return and/or line feed characters so that other input is not
affected.

This page intentionally left blank.

Unit Test

Given the module TEST.BLI (on your disk area and in Attachment #1) which includes trace and monitor macros, insert the appropriate macro calls to test and debug the program. Execute the program and identify the existing bugs.

Note: Do not attempt to debug the program before inserting the debugging code: the routines are small and therefore not difficult to debug by analyzing the source code. For your convenience, the macros

    $TRACE
    $LOOK

are included in the module. Assuming the correct responses are given, the order of execution is:

    in R1 --> in R2 --> in R3 --> out R3 --> out R2 -->
    in R4 --> out R4 --> out R1

This page intentionally left blank.

UTILITIES
MODULE II-10

Course Map
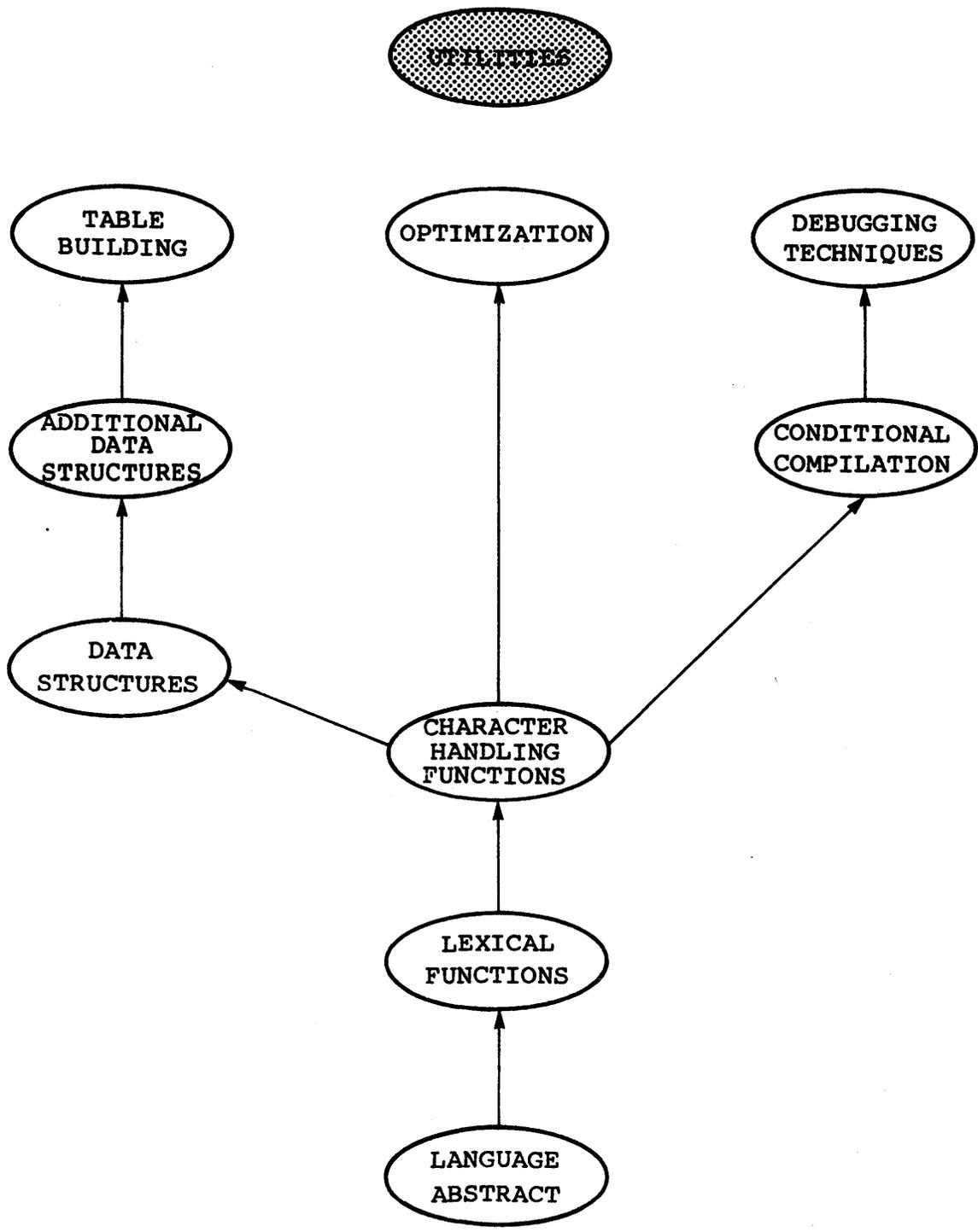
Introduction

This unit presents some of the useful utilities provided for
the BLISS language translators. Included utilities are PRETTY,
BLSCRF, CONDEN, CVT10 and CVT11, and MODULE.

```
┌──────────────────────── Objectives ────────────────────────┐
│                                                             │
│                                                             │
│      This unit exists for the purpose of  acquainting  the  reader  │
│  with a subset of the utilities provided for the BLISS programmer.  │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

```
┌─────────────────────── Sample Test Items ───────────────────┐
│                                                             │
│                                                             │
│      Using the text as necessary, list  at  least  four  utilities  │
│  provided  for  the  BLISS  programmer,  along  with their intended  │
│  usage.                                                     │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

```
┌────────────────────── Additional Resources ─────────────────┐
│                                                             │
│                                                             │
│  VAX-11 BLISS-32 USER'S GUIDE      Chapter 8                │
│                                    Section 2                │
│                                                             │
│  Certain .DOC and .HLP files, including   PRETTY.HLP        │
│                                           BLSCRF.HLP        │
│                                           CVT10.HLP,.DOC    │
│                                           CVT11.HLP,.DOC    │
│                                           MODULE.BLI        │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

PRETTY

PRETTY is a utility program which accepts a BLISS source file
and produces a reformatted source file (and optionally a specially
formatted listing file), using formatting rules as specified in
the VAX-11 Software Engineering Manual. The output file will have
all whitespace (except in strings, macro definitions, comments,
and certain PLIT bodies) removed and replaced. Control
expressions are indented according to hierarchical structure.

To invoke PRETTY, type 'PRETTY'. It will prompt with 'BLF>'
for a file specification line of the form

input.ext [/OUTPUT:name.ext] [/LISTING:name.ext] [/EXIT]

where 'input.ext' is the source file specification. The bracketed
options are available: prompting may be used also. The default
file names are that of the input file. The output file extension
defaults to that of the input; the listing file extension
defaults to '.LST'. (Note: Production of the listing file
doubles PRETTY run time.)

Specific formatting options can be supplied to PRETTY by
means of directives inserted as full-line comments into the source
text. (There are more than 20 such options. For a full list, see
the file, PRETTY.HLP.) Using these options, one can specify
variations in the formatting of pages, macros, PLITs, error
messages, upper and lower case, and synonyms.

Here's an example of PRETTY usage. The file to be PRETTYed
is called UGLY.BLI and looks like this:

```
MODULE UGLY (MAIN = B) = BEGIN ROUTINE B = BEGIN
TTY_PUT_QUO('YUP'); END; END ELUDOM
```

UGLY is syntactically correct; but difficult to read. Now we'll
'PRETTY' it up.

```
@pretty
PRETTY version 6
BLF>ugly.bli/output:nice.bli/exit
MODULE SMALL
ROUTINE B
@ty nice.bli
MODULE SMALL (MAIN = B
                ) =
BEGIN
ROUTINE B =
    BEGIN
```

```
                TTY_PUT_QUO ('YUP');
                END;
        END

        ELUDOM
        @
```

Note that PRETTY tells the user what modules and routines it is in
the process of 'PRETTY'ing up.

By the use of PRETTY, one can tell at a glance if the code
indention properly reflects the program logic. If the 'PRETTY'ed
file has a different indention format than the original file,
perhaps it is due to the program logic being different than
intended. PRETTY is, in this manner, a valuable tool for program
development.

BLSCRF - (BLISS CROSS REFERENCE)

BLSCRF is a program which cross references BLISS source
files. BLSCRF is invoked by the command 'BLSCRF', returning a
single asterisk as a command line prompt. The user may now type a
series of cross reference requests. The command line has the
form:

        outfil,gxrfil=srcfil/switch/switch...

where either "outfil" or ",gxrfil" (but not both) may be omitted
and where all switches are optional. The default file extensions
are

```
        srcfil:  BLI (,B32, B16, B36, B10, B11)
        outfil:  XRF (GXR if /F is specified with a single
                        output file)
        gxrfil:  GXR
        REQUIRE file: REQ (,B32, B16, B36, B10, B11, R32, R16,
                        R36, BLI)
```

Upon completion, if no switches have been specified, outfil
contains a numbered listing of srcfil and its REQUIRE files
followed by a cross reference of srcfil suitable for printing on a
132 column line printer; gxrfil contains a cross reference which
is formatted so it can be sorted with other files of its type to
produce a master cross reference listing of multiple source files.

All REQUIRE files requested in srcfil are included in the cross
reference. All Common BLISS reserved words are excluded from the

cross reference. All line numbers which correspond to lines within REQUIRE files are flagged by a "+" in the cross reference; all routine definitions are flagged ty "*". All symbols directly proceeding the assignment operator are flagged with "#".

In addition, if the BLISS source module has been edited in the style described in the VAX-11 Software Engineering Manual

        (i.e., inserted lines marked by !Aedit as in !AØ5
               modified lines marked by !Medit as in !M11
               deleted lines marked by !Dedit as in !D77)

a list of all lines affected by each edit is output at the end of the cross reference.

For a list of the allowable command switches, see the help file on BLSCRF.

CONDEN

     CONDEN is a program which CONDENses and cleans up the merged and sorted cross reference (see BLSCRF, above).

The following sequence of commands can be used to create a set of GXR (Global X-Reference) files, append them together to form one large GXR file, sort the produced GXR file, and produce the final cross reference:

```
@blscrf
*filea,filea=filea
*fileb,fileb=fileb
*filec,filec=filec
@copy allfil.gxr=filea.gxr,fileb.gxr,filec.gxr
@sort
*files.sor=/record:15Ø/alphanumeric/ascii/key:1:4Ø:a -
        allfil.gxr
@conden
*files.xrf=files.sor
```

FILEA.XRF, FILEB.XRF, and FILEC.XRF now contain the individual cross reference listings. FILES.XRF contains the sorted, merged cross reference of all three.

CVT10 and CVT11 (CONVERT)

CVT10 is a tool for converting BLISS-10 into BLISS-36c.
(CVT11, a sister version, converts BLISS-11 into BLISS-16c.)
CVT10 will do a large percentage of the syntax conversions and a
smaller set of other conversions. CVT10 assumes that the input is
compilable, without errors, by the BLISS-10 compiler. It also
makes no provisions for expanding macros in order to see what was
meant. 'Reasonable' macros and 'reasonable' code will go through
CVT10 unscathed.

To use CVT10, enter the command 'CVT10'. CVT10 will respond by
asking for the names of the input files (separated by spaces).
Next, the names of the output files are requested. (The default
in both cases is TTY:.) When all the files have been converted,
CVT10 will ask for another set of input and output files.

Typically, the input file extension will be .B10 and the output
file extension will be .B36.

For help on using CVT10, see the file, CVT10.HLP. For a full
description of the CVT10 (and CVT11) programs, including
instructions for expanding the range of conversions, see the files
CVT10.DOC and CVT11.DOC.

MODULE

MODULE.BLI is the standard BLISS source module template. It
provides a standard outline for writing BLISS programs. Included
in MODULE.BLI are the full Module headers and Routine headers, as
presented in the VAX-11 Software Engineering Manual.

Exercises

None

This page intentionally left blank.

Unit Test

Using the text as necessary, list at least four utilities
provided for the BLISS programmer, along with their intended
usage.

This page intentionally left blank.

BLISS-32 APPENDIX

This page intentionally left blank.

Introduction

This unit illustrates the steps necessary to execute a
BLISS-32 program on the VAX-11/780 system. Specifically, it
describes the following procedures:

- Getting started on the VAX system
- Editing the source file
- Compiling the BLISS source file
- Linking the resultant OBJ file
- Running the program

Additional Resources

VAX-11 BLISS-32 User's Guide

VAX/VMS Primer

VAX/VMS Text Editing Reference Manual

VAX/VMS LINKER Reference Manual

VAX/VMS Command Language User's Guide

Introduction to VAX-11 RMS

VAX/VMS VAX-11 RMS Reference Manual

VAX/VMS Programmer's Guide to Debugging

VAX-11 Common Run-Time Procedure Library Reference Manual

VAX/VMS System Services Reference Manual

VAX/VMS Symbolic Debugging Reference Manual

This unit teaches you how to transform a BLISS-32 source program into a running program on the VAX-11/780 system.

There are five steps involved in the process:

- Get logged on to the VAX system
- Write a BLISS-32 source program using EDIT
- Compile the source program by invoking the BLISS-32 compiler
- Link the resultant object module using LINK, yielding an executable image of your program
- Run the image on the VAX system

GENERAL PROCEDURE:

STEP 1: Get LOGGED on to the system.

First, see the system manager to arrange an account with a user-name and password.

Let's assume that you've found a terminal that's connected to a VAX-11/780 system. Get the system's attention by hitting RETURN. The VAX system will respond with

USER-NAME:

Enter the user-name and hit RETURN. Next, the system will request the password

PASSWORD:

Enter your password and hit RETURN. If your USER-NAME and PASSWORD are correct, the VAX system will display a greeting like this:

WELCOME TO VAX/VMS VERSION 1.00
$

The '$' is the VAX/VMS system prompt.

You are now logged on to the system. If you have any question about the commands which you can give VAX, try typing 'help'. You'll get a list of subjects for which the system has a HELP message.

For example, if you want help on the DIRECTORY command:

```
$ help directory
DIRECTORY
        Provides a list of files or info about
        a file or group of files.
        Format:  DIRECTORY [file-spec,...]
        Additional info available: Parameters
                Qualifiers
```

Note the 'Additional info' list. This is a time saving feature. By giving more information only when you ask for it, you can zero in on just the help you need.

Say you want more help on the DIRECTORY command qualifiers:

```
$ help dir qua
DIRECTORY
    QUALIFIERS
        /BRIEF
          Lists only the file name, filetype,
          and file version of each file to
          be listed.
        /FULL
          ...
          .
          .
          .
```

Note that you only need to type enough of a command to uniquely specify it, as in 'hel dir qua'.

If you wish to see the present time:

```
$ show time
   15-JAN-78 17:34:23
```

There are several other things the VAX system can 'show'. Find out which ones by getting HELP on SHOW.

When you are finished using the system, use the LOGOUT command to end the terminal session:

```
$ logout
```

For more information on using the VAX system, see: The VAX/VMS Primer.

STEP 2:   Create a BLISS-32 source program using EDIT.

The editor is invoked to create or modify a source file
with a command like this:

        $ edit test.b32

If the file already exists, the EDITor retrieves it and
prepares it for your modifications.   If no such file
exists, the EDITor creates it for you.

Note: 'B32' is the proper file type for a BLISS-32 source
file.

        An editing session now begins, during which you will
direct the EDITor using certain special commands. The
default EDITor for the VAX system is called 'SOS'.   Its
commands are nearly identical with those of the SOS
editors on other DEC computer systems.

        SOS is a line-oriented editor; that is, it 'sees'
text in terms of lines. A line is a string of characters
and spaces. Every line in an SOS file has a line number
associated with it.   When SOS displays a line, it is
preceeded by its line number:

00100 This is a sample line of text.

        SOS responds to the EDIT command by displaying its
version number and the full file specification. SOS then
prompts you to begin entering lines:

        $ edit test.b32
        SOS   V02.04A
        INPUT:DB0:[300,21]TEST.B32;1
        00100

The line number prompt, '00100', indicates that you are in
INPUT MODE.   In this mode, each line you enter is placed
into the file.

        Terminate each line by hitting RETURN.   SOS
automatically increments the line number by 100 and
prompts you for more input.

        When you are finished inputting lines, hit the ESCAPE
(<esc>) key to get out of INPUT MODE. (On some terminals,
the equivalent key is labeled 'ALTmode', 'SELect', or
'PREfix'.) The <esc> key is echoed to your terminal as a

dollar sign ($). Upon the echo of the $, you are in the
EDIT MODE. In this mode, SOS interprets each line you
enter as one of its special commands. The EDIT MODE
prompt is a '*'.

When you are done with the file, you can exit the
EDITor by entering an 'E' at the EDIT MODE level. SOS
will write the entered lines into a disc file under the
specified name and return you to VAX command level.

Summary of Frequently Used SOS Commands:

Command	Arguments*	Function
<RET>	None	Print the next line in the file
<ESC>	None	Print the previous line
P	position or range	Display line(s)
I	position	Insert new line(s) into file
N	increment and/or range	Renumber the lines in file
R	position or range	Replace one or more lines with new line(s)
D	position or range	Delete line(s) from the file
F	string<ESC>	Find and print the next line containing the specified string

*key:
    position   means you can specify a single line
                  number
    range   means you can specify a range of line
                  numbers of the form <firstline>:
                  <lastline>
    increment is a numeric value for line number
                  incrementing
    string   is any string of characters

For more information on the SOS Text EDITor, see:
VAX/VMS Text Editing Reference Manual.

STEP 3:   Compile the program.

The BLISS-32 source file that you've created must   now   be
compiled.   The   compiler   checks   your source program for
syntax and programming errors, then translates this   input
source   file   into   a   binary   form.   The translated code,
called the 'object module', is   written   by   the   compiler
into a file called the 'object module file', which has the
file type, 'OBJ'.

You can invoke the BLISS-32 compiler   by   entering   a
command string of the form:

BLISS [<qualifiers>] <sourcefile>[,<sourcefile>,...]

Source file names   may   be   separated   by   a   comma.   The
following qualifiers are supported:

QUALIFIER	DEFAULT
/[NO]LIST: [<filespec>]	/NOLIST
/[NO]OBJECT: [<filespec>]	/OBJECT
/[NO]LIBRARY: [<filespec>]	/NOLIBRARY
/[NO]DEBUG	/NODEBUG
/[NO]QUICK	/NOQUICK
/[NO]CODE	/CODE
/VARIANT[=<n>]	/VARIANT=0
/TERMINAL:(	/TE:(ERR,NOSTAT)
[NO]ERRORS,	
[NO]STATISTICS)	
/SOURCE:(	/SOURCE:(HEAD,PAGE=55,
[NO]HEADER	NOLIB,NOREQ,SOURCE,
PAGE_SIZE=<n>	NOEXP,NOTRACE)
[NO]LIBRARY	
[NO]REQUIRE	
[NO]SOURCE	
[NO]EXPAND_MACROS	
[NO]TRACE_MACROS)	
/OPTIMIZE:(	/OPT:(LEVEL=2,SPACE,SAFE)
LEVEL=<n>	
SPACE | SPEED	
[NO]SAFE)	
/MACHINE_CODE:(	/MACH:(NOASS,SYM,BIN,COM,
[NO]ASSEMBLER	NOUNIQUE)
[NO]SYMBOLIC	
[NO]COMMENTARY	
[NO]BINARY	
[NO]UNIQUE_NAMES)	

If no <filespec> is supplied in LIST, OBJECT or LIBRARY
qualifiers, default is to the name of the first input file
(with the appropriate file type, i.e., LIS, OBJ, L32).

The OBJECT and LIBRARY qualifiers are mutually exclusive.

(See APPENDIX A for a detailed description of each
qualifier.)

STEP 4: 'Link the object module.

The OBJECT module produced by the BLISS-32 compiler is not
in itself, executable: generally, an object module
contains references to other programs or routines that
must be bound with the object module before it can be
executed. This is the function of the LINKer.

The LINKer is invoked by a command of the form:

LINK   [/<command   qualifiers>]   <filespec(s)>[<file
qualifiers>]

</command qualifiers> specify output file options

<filespec(s)> specify the input OBJect file(s) to be
linked (Default filetype in <filespec> is OBJ.)

</file qualifiers> specify input file options

    Note that the LINK command can be entered without an
accompanying file specification. The system responds with
the prompting message:

    $_FILE:

You should type the file specification on the same line as
the prompting message.

    Multiple file specifications can be entered, each
separated from the preceding specification by a comma. A
single executable image is created from the input files
specified. If no output file is specified, the LINKer
produces an executable image with the same name as the
first object module, and the type EXE. (No executable
image is produced if the /NOEXECUTE qualifier is
specified.) The following table lists some of the most
often used LINKer qualifiers.

LINKer Qualifiers

Command qualifiers	Default
/[NO]BRIEF	/NOBRIEF
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG[:<filespec>]	/NODEBUG
/[NO]EXECUTABLE[=<filespec>]	/EXECUTABLE
/[NO]FULL	/NOFULL
/[NO]MAP[=<filespec>]	/NOMAP

File qualifiers	Default
/[NO]INCLUDE[=<module-name>[,...]]	/NOINCLUDE
/[NO]LIBRARY	/NOLIBRARY

LINKer command qualifiers affect the output produced by the LINKer. By including the appropriate qualifiers, you can determine the type of executable image produced (if any), and the type of map file produced (if any).

(See APPENDIX B for a detailed description of each qualifier.)

STEP 5: Run the program.

The LINKer has created an executable image of your program. You can now run the program by the use of the following general command string:

    RUN [<qualifiers>]<filespec>

If the <filespec> does not include a file type, the RUN command assumes a file type of EXE.

Normally, no <qualifiers> are used in the RUNning of a program. However, by the use of the proper <qualifiers>, you can influence such RUN factors as:

    o   delay time
    o   interval time
    o   priority
    o   privileges
    o   process-name
    o   swapping
    o   user identification code

For more information, see:

VAX-11 Common Run-Time Procedure Library Reference Manual,
VAX/VMS System Services Reference Manual, or
VAX/VMS Command Language User's Guide.

AN EXAMPLE OF THE WHOLE PROCEDURE:

Suppose that you wanted a BLISS program that looked like this:

```
MODULE SKELETON (MAIN = MAINLOOP) =
BEGIN
!++
!    This is a test BLISS program
!--
REQUIRE 'TUTIO';
ROUTINE MAINLOOP: NOVALUE =
    BEGIN
    TTY_PUT_QUO('WHAT HO !');
    END;
END
ELUDOM
```

Let's look at the steps necessary to take this program from
paper to an executing program on the VAX-11/780 system.

STEP 1: Log on to the system.

Make sure the terminal is powered ON. Hit RETURN and
enter USER-NAME and PASSWORD when VAX requests them. The
system will respond with a message like this:

```
WELCOME TO VAX/VMS VERSION 1.00
$
```

Note the system prompt ($) ending the message.

STEP 2: Enter the BLISS-32 source program into a file.

We will call our program something clever, like
'TEST.B32'. Here's the way the editing session would go.

```
$ edit test.b32
SOS    V02.04A
INPUT:DB0:[300,21]TEST.B32;1
00100    MODULE SKELETON (MAIN = MAINLOOP) =
00200    BEGIN
00300    !++
```

```
00400    !     This is a BLISS test program.
00500    !--
00600    REQUIRE 'TUTIO';
00700    ROUTINE MAINLOOP: NOVALUE =
00800        BEGIN
00900        TTY_PUT_QUO('WHAT HO !');
01000        END;
01100    END
01200    ELUDOM$
```

The $ in the last line is the echoed <ESC> that you hit to
go from INPUT MODE to the EDIT MODE. SOS responds with

*

The file looks OK, so exit SOS by entering 'E':

```
*E
[DB0:[300,21]TEST.B32;1]
$
```

Note that SOS displays the full file specification.

STEP 3: Compile the program.

   Now you are ready to invoke the BLISS-32 compiler. Let's
   say, as an option, that you want a listing of the compile.
   The command string to enter is:

      BLISS /LIST TEST

   BLISS-32 will compile TEST.B32. If the compilation is
   successful, the produced OBJECT module is named TEST.OBJ,
   and the compiler listing is named TEST.LIS.

STEP 4: Link the object module.

   Invoke the LINKer with the command:

      LINK TEST

   If the link is successful, it will produce the file named
   TEST.EXE.

STEP 5:   Run the program.

You can now run your program by entering

    RUN TEST

APPENDIX A

Compilation Qualifiers

[NO]LIST: [<filespec>]
    Specifies that the listing file is to be <filespec>.

[NO]OBJECT: [<filespec>]
    Specifies that the object code is to be placed in
<filespec>.

[NO]DEBUG
    Indicates whether or not the compiler should produce a
symbol table that may be used with the debugger.

[NO]QUICK
    Requests a faster, non-optimized compilation.   (The lack
of optimization may make it easier to relate the source code
to the generated code.)

[NO]CODE
    Specifies whether or not the compiler should produce
object code.   You might use /NOCODE to simply perform syntax
checking of the source program; because the compiler does not
produce code, the compilation time is reduced.

VARIANT[=<n>]
    Specifies the value of the predeclared literal %VARIANT.
If no value is specified for <n>, it defaults to a value of 1;
otherwise %VARIANT has the specified value.   If /VARIANT is
not specified at all, %VARIANT has a value of 0.

TERMINAL:([NO]ERRORS,[NO]STATISTICS)
    Controls whether or not ERROR and STATISTICal information
are displayed on the user terminal during the compilation. If
STAT is specified, the compiler will display the names and
sizes of all routines that are being compiled.

SOURCE:(<spec>,<spec>,<spec>,...)
    Specifies what information is to appear in the listing
file.
    [NO]HEADER
        Indicates whether the compiler should print normal
        page header information at the top of each page of the
        listing file. If NOHEADER is specified, the page
        header is supressed, as well as the form feed and any
        compiler space dependent summary information.
    PAGE_SIZE=<n> Specifies the number of lines in a page in
        the listing file.

[NO]REQUIRE Controls whether or not the contents of all the
require files be included in the listing.

[NO]SOURCE Indicates whether or not the BLISS-32 source
statements will appear in the listing.

[NO]EXPAND_MACRO Specifies whether or not all MACROs shall
be expanded wherever they appear in the source
listing.

[NO]TRACE_MACROS Requests that the listing include a trace
of MACRO expansions.

OPTIMIZE:(LEVEL=<n>,SPACE | SPEED,[NO]SAFE)
Indicates whether the compiler should optimize code across
mark points. Optimization can be set for either SPACE
efficiency or time efficiency (SPEED). [NO]SAFE indicates
whether or not named variables in the source code will be
addressed only by name. Use NOSAFE to indicate that variables
can be addressed by pointers, and not just by name.

MACHINE_CODE:(<spec>,<spec>,<spec>,...)
The MACHINE_CODE qualifier requests a listing of the
generated object code, the format of which is determined by
the following subqualifiers:

[NO]ASSEMBLER Specifies whether or not a listing of the
object code is to be produced in a suitable format for
assembly.

[NO]SYMBOLIC Specifies whether or not a listing of the
object code is to be produced in a suitable format for
symbolic interpretation by the programmer.

[NO]COMMENTARY Specifies whether or not a listing of the
object code is to be produced with a compiler
commentary.

[NO]BINARY Specifies whether or not a listing of the object
code in binary format is to be produced.

[NO]UNIQUE_NAMES Indicates whether the compiler should
produce unique names for OWN variables and non-global
routine names when it creates a listing that is to be
assembled.

APPENDIX B

LINKer Qualifiers

[NO]BRIEF

Requests the LINKer to produce a brief map (memory allocation) file. The /BRIEF qualifier is valid only if /MAP is also specified; and it must follow the /MAP qualifier in the command line.

The /BRIEF listing contains:
o A summary of the image characteristics
o A list of all object modules included in the image
o A summary of the LINK-time performance statistics

(The effect of the /BRIEF qualifier is to withhold the summary of global symbols from the memory allocation listing.)

[NO]CROSS_REFERENCE

Controls whether or not the memory allocation listing (map) contains a symbol cross reference. The /CROSS_REFERENCE qualifier is valid only if /MAP is also specified; and it must follow the /MAP qualifier in the command line. A symbol cross reference lists each global symbol referenced in the image, its value, and all modules in the image that refer to it.

[NO]DEBUG[=<filespec>]

Controls whether or not an executable image is bound with a debugger. /DEBUG is valid only if /EXECUTABLE is specified, either explicitly or by default. The /DEBUG qualifier optionally accepts the name of an alternate, user-specified debugger. If a file specification is entered, and it does not contain a file type, the LINKer assumes the default file type of OBJ. If /DEBUG is specified without a file specification, the default VAX/VMS Debugger, DEBUG, is linked with the image. For more information on DEBUG, see the VAX/VMS Programmer's Guide to Debugging.

[NO]EXECUTABLE[=<filespec>]

Controls whether or not the LINKer produces an executable image and optionally provides a file specification for the output image file. By default, the LINKer creates an executable image with the same file specification as the first input file specified and a file type of EXE.

Use /NOEXECUTABLE when you want to determine the outcome of linking a set of modules, without incurring the LINKer overhead required to create an image file.

[NO]FULL
     Requests the LINKer to produce a full map (memory
allocation) listing.   The /FULL qualifier is valid only if
/MAP is specified, and must follow the /MAP qualifier in the
command line.
     A /FULL listing contains the following information:
     o A summary of the image characteristics
     o A list of all object modules included in the image
     o A summary of LINK-time performance statistics
     o A list of global symbols
     o Detailed descriptions of each program section in the
image file

[NO]MAP[=<filespec>]
     Controls whether or not a memory allocation listing (map)
is produced and optionally defines the file specification. If
/MAP is specified, the qualifiers /BRIEF, /FULL, or
CROSS_REFERENCE may also be specified to control the contents
of the map. If none of these qualifiers if specified, then
the map will contain:
     o A summary of the image characteristics
     o A list of all object modules included in the image
    .o A summary of LINK-time performance statistics
     o A summary of global symbols
     If /MAP is specified without an output file specification,
the output file is given the same file name as the first input
file specified and a file type of MAP.

[NO]SHAREABLE[=<filespec>]
     Requests the LINKer to produce a shareable image file
rather than an executable image. If no file specification is
given, the LINKer provides the image file with the same file
name as the first input file and a file type of EXE.
     If /SHAREABLE is specified, /DEBUG may not be specified.
Shareable images cannot be run with the RUN command; however,
they may be linked with object modules to produce executable
images.   (See the /SHAREABLE qualifier in the information on
file qualifiers, below.)

[NO]SYMBOL_TABLE[=<filespec>]
     Requests the LINKer to create a separate file in object
module format containing symbol definitions for all symbols
contained in the image. If /SYMBOL_TABLE is specified without
a file specification, the LINKer creates a file with the same
file name as the image file and a file sype of STB. If /DEBUG
is specified, the LINKer includes the symbol definitions in
the image for use by the debugger, and also creates a separate
symbol table file.
     The symbol table file can be used as input to subsequent

LINK commands, to provide the symbol definitions to other images.

File qualifiers can set certain options for the input file(s).

[NO]INCLUDE[=<module-name>[,...]]
/NOINCLUDE indicates that the associated input file is an object module library, and that only the module names specified are to be unconditionally included as input to the LINKer. If /INCLUDE is specified, /LIBRARY can also be specified, to indicate that the same library should also be used to search for unresolved references. At least one <module-name> must be specified. If more than one is specified, they must be enclosed in parentheses and separated by commas.

[NO]LIBRARY
/LIBRARY indicates that the associated input file is an object module library, which is to be searched for modules that resolve any undefined symbols in the input file(s).
If the associated input file specification does not include a file type, the LINKer assumes the default file type of LIB.
A library may not be specified as the first input file unless the /INCLUDE qualifier is also specified to indicate which modules in the library are to be included in the input. You can use both /INCLUDE and /LIBRARY to qualify a file specification; in that case, the explicit inclusion of modules occurs first, then the library is used to search for any unresolved references.

[NO]SHAREABLE
/SHAREABLE indicates whether the associated input file is a previously-linked shareable image. If /SHAREABLE is specified the associated input file specification does not include a file type; the LINKer assumes the default file type of EXE.

This page intentionally left blank.

BLISS-36 APPENDIX

This page intentionally left blank.

Introduction

This unit illustrates the steps necessary to execute a BLISS-36 program on the DECSYSTEM-10/20. Specifically, it describes the following procedures:

- Getting started on the DECSYSTEM-10/20
- Editing the source file
- Compiling the BLISS source file
- Linking the resultant REL file
- Running the program

Additional Resources

Getting Started with the DECSYSTEM-20

DECSYSTEM-10 Software Notebooks (13 Volumes)

DECSYSTEM-20 Edit User's Guide

DECSYSTEM-20 User's Guide

DECSYSTEM-20 LINK Reference Manual

DECSYSTEM-10 Operating System Command Manual

DECSYSTEM-20 EDIT Reference Manual

DECSYSTEM-20 EDIT Reference Card

VAX-11 BLISS-32 User's Guide

This unit teaches you how to transform a BLISS-36 source program into a running program on the DECSYSTEM-10/20. For the sake of simplicity, all examples will be given from the point of view of TOPS-20, the monitor that runs on the DECSYSTEM-20. (For more information on TOPS-10, see the appropriate references as mentioned in ADDITIONAL RESOURCES.)

There are five steps involved in the process:

- Get logged on to the DECSYSTEM-10/20
- Write a BLISS-36 source program using EDIT
- Compile the source program by invoking the BLISS-36 compiler
- Link the resultant object module, load it into core, and save the executable image of your program
- Run the image on the DECSYSTEM-10/20 system

GENERAL PROCEDURE:

    STEP 1: Get LOGGED on to the system.

        First, see the system manager to arrange an account with a user-name and password (and possibly an account number).

        Let's assume that you've found a terminal that's connected to a DECSYSTEM-10/20. Get the system's attention by hitting CTRL-C (that's the <CTRL> and C keys simultaneously). The DECSYSTEM-20 will respond with a message like this:

            EDUCATIONAL SERVICES , TOPS-20 Monitor 3(1371)
            @

        The '@' is the TOPS-20 system prompt. It says that you are at 'command level'. Now type the following information:

            LOGIN <user-name> <password> [<account>]

        If your user-name, password, (and account number, if applicable) are correct, TOPS-20 will display a message like this:

            Job 9 on TTY31 13-Jun-78 10:23:42
            @

        You are now logged on to the system. If you have any question about the commands which you can give TOPS-20,

try typing 'help *'. You'll get a list of subjects for
which the system has a HELP message.

For example, if you want help on BLISS:

    @help bliss
    BLISS-36 is the new BLISS compiler (BLISS.EXE) which
    runs under TOPS-10 or TOPS-20 and generates code for
    KA, KI, and KL processors. The BLISS system is
    '.
    .
    .

If you wish to see the present time:

    @daytime
     Friday, October 13, 1978 11:34:32

You can gather a great deal of information about the
TOPS-20 commands using the 'HELP *' facility.

When you are finished using the system, use the LOGOUT
command to end the terminal session:

    @logout
    Killed Job 9, User WITHROW, Account 234, TTY 31,
        at 15-Jun-78 20:23:42, Used 0:3:13 in 0:23:34

For more information on using the DECSYSTEM-20 system,
see: The DECSYSTEM-20 User's Guide.

STEP 2:   Create a BLISS-36 source program using EDIT.

The editor is invoked to create or modify a source file
with a command like this:

    $ edit test.bli

If the file already exists, the EDITor retrieves it and
prepares it for your modifications.   If no such file
exists, the EDITor creates it for you.

Note that 'Bli' (and 'b36') are the two default file types
for BLISS-36.

    An editing session now begins, during which you will
direct the EDITor using certain special commands. The
default EDITor for the DECSYSTEM-20 system is called
'SOS'.   Its commands are nearly identical with those of
the SOS editors on other DEC computer systems.

    SOS is a line-oriented editor; that is, it 'sees'
text in terms of lines. A line is a string of characters
and spaces. Every line in an SOS file has a line number
associated with it.   When SOS displays a line, it is
preceeded by its line number:

    00100 This is a sample line of text.

    SOS responds to the EDIT command by displaying its
version number and the full file specification. SOS then
prompts you to begin entering lines:

    @edit test.bli

    %File not found, Creating New file
    Input: TEST.BLI.1
    00100

The line number prompt, '00100', indicates that you are in
INPUT MODE.   In this mode, each line you enter is placed
into the file.

    Terminate each line   by   hitting   RETURN.   SOS
automatically increments the line number by 100 and
prompts you for more input.

    When you are finished inputting lines, hit the ESCAPE
(<esc>) key to get out of INPUT MODE. (On some terminals,
the equivalent key is labeled 'ALTmode', 'SELect', or

'PREfix'.) The <esc> key is echoed to your terminal as a
dollar sign ($). Upon the echo of the $, you are in the
EDIT MODE. In this mode, SOS interprets each line you
enter as one of its special commands. The EDIT MODE
prompt is a '*'.

When you are done with the file, you can exit the
EDITor by entering an 'E' at the EDIT MODE level. SOS
will write the entered lines into a disc file under the
specified name and return you to TOPS-20 command level.

Summary of Frequently Used SOS Commands:

Command	Arguments*	Function
<RET>	None	Print the next line in the file
<ESC>	None	Print the previous line
P	position or range	Display line(s)
I	position	Insert new line(s) into file
N	increment and/or range	Renumber the lines in file
R	position or range	Replace one or more lines with new line(s)
D	position or range	Delete line(s) from the file
F	string<ESC>	Find and print the next line containing the specified string

*key:
   position   means you can specify a single line
               number
   range   means you can specify a range of line
               numbers of the form <firstline>:
               <lastline>
   increment   is a numeric value for line number
               incrementing
   string   is any string of characters

For more information on the SOS Text EDITor, see:
DECSYSTEM-20 EDIT User's Guide

STEP 3:   Compile the program.

The BLISS-36 source file that you've created must now be
compiled.   The compiler checks your source program for
syntax and programming errors, then translates this input
source file into a binary form. The translated code,
called the 'object module', is written by the compiler
into a file called the 'object module file', which has the
file type, 'REL'.

You can invoke the BLISS-36 compiler by typing simply
'BLISS'.   The BLISS-36 compiler will next display its
prompt:

BLISS>

You should now enter the command line, which, in its
simplest form contains one filename, like this:

BLISS>file

This will cause FILE.B36 (or FILE.BLI) to be compiled, and
the FILE.REL to be created.

A number of switches can also be specified on the command
line.   (For a full list of switches, see Appendix A.)

STEP 4:   Link the object module.

The OBJECT module produced by the BLISS-36 compiler is not
in   itself, executable: generally, an object module
contains references to other programs or routines that
must be bound with the object module before it can be
executed.   This is the function of the linker.

The linker is invoked by a command of the form:

LOAD file

(Default filetype is REL.)

The LOAD command will not only invoke the linker;   it
will also load the linked module into core. At this
point, the easiest way to start the program running is to
enter the command 'START'.   However, if you wish to run
the program in the future, you can avoid the repeated
LOADing and STARTing by now SAVING the core image of the
LOADed program. To do this enter the command:

        @save file.exe

The executable core image is now permanently SAVEd in the
file with extension, 'EXE'.

For more information, see:

DECSYSTEM-20 LINK Reference Manual

STEP 5: Run the program.

    The SAVEd core image file can now be executed, by entering
    the command:

        @run file.exe

    If the <filespec> does not include a file type, the RUN
    command assumes a file type of EXE. For more information,
    see:

    DECSYSTEM-20 User's Guide

AN EXAMPLE OF THE WHOLE PROCEDURE:

    Suppose that you wanted a BLISS program that looked like this:

```
        MODULE SKELETON (MAIN = MAINLOOP) =
        BEGIN
        !++
        !    This is a test BLISS program
        !--
        REQUIRE 'TUTIO';
        ROUTINE MAINLOOP: NOVALUE =
            BEGIN
            TTY_PUT_QUO ('I AM FLAT!');
            END;
        END
        ELUDOM
```

    Let's look at the steps necessary to take this program from
        paper to an executing program on the DECSYSTEM-20.

STEP 1: Log on to the system.

    Make sure the terminal is powered ON. Hit CTRL-C.   Let's
    say that USER-NAME, PASSWORD, and ACCOUNT are 'WITHROW',

'PAL', and '432', respectively. In response to the opening message of TOPS-20 enter this line:

@login withrow pal 432

Note that TOPS-20 will not echo the characters of the password (for the sake of security.)

The system will respond with a message like this:

```
Job 15 on TTY52 14-Aug-78 00:34:22
@
```

Note the system prompt (@) ending the message.

STEP 2: Enter the BLISS-36 source program into a file.

We will call the program 'FLAT.BLI'.
Here's the way the editing session would go.

```
@edit flat.bli
%File not found, Creating New file
Input: FLAT.BLI.1
00100    MODULE SKELETON (MAIN = MAINLOOP) =
00200    BEGIN
00300    !++
00400    !    This is a BLISS test program.
00500    !--
00600    REQUIRE 'TUTIO';
00700    ROUTINE MAINLOOP: NOVALUE =
00800        BEGIN
00900        TTY_PUT_QUO('I AM FLAT!');
01000        END;
01100    END
01200    ELUDOM$
```

The $ in the last line is the echoed <ESC> that you hit to go from INPUT MODE to the EDIT MODE. SOS responds with

```
*
```

The file looks OK, so exit SOS by entering 'E':

```
*E
[FLAT.BLI.1]
@
```

STEP 3: Compile the program.

Now you are ready to invoke the BLISS-36 compiler. Let's say, as options, that you want a listing of the compile, statistics to be displayed, and to return to monitor level upon completion of the compile. The command strings to enter are:

    @bliss
    BLISS>flat/list/stat/exit

BLISS-36 will compile FLAT.BLI. If the compilation is successful, the produced object module is named FLAT.REL, and the compiler listing is named FLAT.LIS.

STEP 4: Link the object module.

Invoke the Linker with the command:

    @load flat

If the link/load is successful, it will generate the message 'EXIT', and return to monitor level. You can then save the core image with the command:

    @save flat

There now exists a new file, 'FLAT.EXE', which can be executed.

STEP 5: Run the program.

You can now run your program by entering

    RUN FLAT

APPENDIX A

BLISS-36 Compiler Switches

A listing of the available switches is given below.
Defaults are marked with an asterisk(*).

SWITCH   DESCRIPTION

/C(ODE)*   Generate object code.

/D(EBUG)   Generate symbols for debugging, emit debug
                        linkages, and disable certain optimizations
                        so the debugger doesn't get confused.

/ER(RS)*   Report errors to the terminal.

/EXI(T)   Exit back to EXEC after compilation.

/EXT(ENDED)   Program to run in a non-zero section on a
                        Model B processor.

/F(ORMAT):   Specify certain options to control the format
                        of the listing file (for full list of FORMAT
                             switches, see below.)

/H(EADER)*   Produce a heading on the top of each page of
                        the listing file including configuration
                        information.

/KA(10)   Assume generated code is to be executed on a
                        KA-10.

/KI(10)   Assume generated code is to be executed on a
                        KI-10.

/KL(10)*   Assume generated code is to be executed on a
                        KL-10 or KL-20.

/LIB(RARY)   Interpret the source file(s) as a library
                        source file. The resultant object file
                        (.L36) is the precompiled library file that
                        can be requested from a source program with a
                        library-declaration. A library file can only
                        be requested by a compiler if that same
                        compiler was the one that built the library.

/LIS(TING):fs	Produce a listing file. If no filespec (fs) is given as an argument, use the default extension .LST.
/NOxxx	Invert the meaning of switch /xxx.
/OPTI(MIZE)*	Perform full flow analysis.
/OPTL(EVEL):n	Specify the degree of optimization. N must be in the range 0 to 3. Request 3 for maximum optimization. Default is 2.
/P(AGSIZ):n	Specify the number of lines printed on each page of the listing file. N must be in the range 20 to 52. Default is 52.
/Q(UICK)	Requests a "quick" compilation, possibly at the expense of some optimization.
/SA(FE)*	Specifies that the source program is coded in a "safe" manner, i.e. values of named variables are changed only by explicit assignment to the named variable. If /NOSAFE is specified, the compiler assumes that indirect assignments invalidate a larger class of variables.
/ST(ATISTICS)	Produce routine names on the terminal as they are compiled.
/TOPS1(0)	Produce code to run under TOPS-10.
/TOPS2(0)*	Produce code to run under TOPS-20.
/U(NAMES)	Produce unique names for own variables and non-global routine names in the listing file when it is to be assembled.
/V(ARIANT):n	Assign n to be the value of the lexical function %VARIANT. If not present (default), zero is assumed. If present and no value is given, 1 is assumed.
/Z(IP)	Optimized for time over space if there is a choice to be made.

The following are the options that can be given with the /FORMAT switch. The form of the /FORMAT switch is given by:

/FORMAT:option

or

/FORMAT:(option,option, ...)

/A(SSEMBLY)   Produce a listing file that can be assembled. (Some modification may be necessary.)

/B(INARY)*   Include in the listing file the binary code generated.

/C(OMMENTARY)*   Include in the listing file commentary on the operands of each instruction. Currently, this consists of a source line number.

/E(XPAND)   Include the expansion of each macro invocation.

/L(IBRARY)   Include a trace identifying the library accessed by each library-declaration and the first use of each name whose definition is obtained from a library file.

/NOxxx   Invert the meaning of option xxx.

/O(BJECT)*   List the compiled code. The ASSEMBLY, SYMBOLIC, BINARY, and COMMENTARY options determine the format of the compiled code.

/R(EQUIRE)   Do not modify the listing control counter when opening or closing files specified in a REQUIRE declaration. In the default case, this results in not listing each REQUIRE file. See SOURCE.

/SO(URCE)*   Increments the listing control counter. The initial value of the counter is 1. When the counter is greater than 0, source is listed. Require-declarations automatically decrement the counter for the length of the require file unless REQUIRE is set.

/SY(MBOLIC)*   List the instructions generated, using as many program symbols as possible.

/T(RACE)   Trace macro expansion. This includes the

resulting   stream   of   lexemes   produced   by
EXPAND.

BLISS-16 APPENDIX

This page intentionally left blank.

Introduction

This unit illustrates the steps necessary to translate a
BLISS-16 program on the DECSYSTEM-10/20. Specifically, it
describes the following procedures:

- Getting started on the DECSYSTEM-10/20
- Editing the source file
- Compiling the BLISS source file

```
┌──────────────────── Additional Resources ────────────────────┐
│                                                               │
│                                                               │
│   Getting Started with the DECSYSTEM-20                        │
│                                                               │
│   DECSYSTEM-10 Software Notebooks (13 Volumes)                │
│                                                               │
│   DECSYSTEM-20 Edit User's Guide                              │
│                                                               │
│   DECSYSTEM-20 User's Guide                                   │
│                                                               │
│   DECSYSTEM-10 Operating System Command Manual               │
│                                                               │
│   DECSYSTEM-20 EDIT Reference Manual                         │
│                                                               │
│   VAX-11 BLISS-32 User's Guide                               │
│                                                               │
│   DECSYSTEM-20 EDIT Reference Card                           │
│                                                               │
│   various documentation for the PDP-11 systems               │
│                                                               │
└───────────────────────────────────────────────────────────────┘
```

This unit will show you how to transform a BLISS-16 source program into an object file ready to be transported to a PDP-11 system and run there. The translator system for BLISS-16 is actually a pre-processor chaining to a compiler chaining to a cross-assembler. The whole system is called BLISS-16c ('c' for cross-compiler.) Most of that operation is performed without the need of user initiation. This unit will assume that the user is new to the use of the DECSYSTEM-10/20, and will, therefore, explain the generation of the final object file in detail.

For the sake of simplicity, all examples will be given from the point of view of TOPS-20, the monitor that runs on the DECSYSTEM-20. (For more information on TOPS-10, see the appropriate references as mentioned in ADDITIONAL RESOURCES.)

There are four steps involved in the process:

- Get logged on to the DECSYSTEM-10/20
- Write a BLISS-16 source program using EDIT
- Compile the source program by invoking the BLISS-16c translator package
- Transport the resultant object file to a PDP-11 system for task-building and execution

GENERAL PROCEDURE:

STEP 1: Get LOGGED on to the system.

First, see the system manager to arrange an account with a user-name and password (and possibly an account number).

Let's assume that you've found a terminal that's connected to a DECSYSTEM-10/20. Get the system's attention by hitting CTRL-C (that's the <CTRL> and C keys simultaneously). The DECSYSTEM-20 will respond with a message like this:

    EDUCATIONAL SERVICES , TOPS-20 Monitor 3(1371)
    @

The '@' is the TOPS-20 system prompt. It says that you are at 'command level'. Now type the following information:

    LOGIN <user-name> <password> [<account>]

If your user-name, password, (and account number, if
applicable) are correct, TOPS-20 will display a message
like this:

        Job 9 on TTY31 13-Jun-78 10:23:42
        @

You are now logged on to the system.   If you have any
question about the commands which you can give TOPS-20,
try typing 'help *'. You'll get a list of subjects for
which the system has a HELP message.

For example, if you want help on BLISS-16c:

        @help bls16c
        The BLISS-16c is a translator and compiler in one
        package: the translator produces BLISS-11 code
        from BLISS-16c source code; the compiler is in
        fact the BLISS-11 compiler.
        .
        .
        .

If you wish to see the present time:

        @daytime
        Friday, October 13, 1978 11:34:32

You can gather a great deal of information about the
TOPS-20 commands using the HELP * facility.

When you are finished using the system, use the LOGOUT
command to end the terminal session:

        @logout
        Killed Job 9, User PEGRAM, Account 234, TTY 31,
          at 15-Jun-78 20:23:42, Used 0:3:13 in 0:23:34

For more information on using the DECSYSTEM-20 system,
see: The DECSYSTEM-20 User's Guide.

STEP 2:   Create a BLISS-16 source program using EDIT.

The editor is invoked to create or modify a   source   file
with a command like this:

    $ edit test.b16

If the file already exists, the EDITor   retrieves   it   and
prepares   it   for   your   modifications.   If   no such file
exists, the EDITor creates it for you.

Note: 'b16' is the default file type for BLISS-16.

An editing session now begins, during which you   will
direct   the   EDITor   using   certain special commands.   The
default EDITor   for   the   DECSYSTEM-20   system   is   called
'SOS'.   Its   commands   are nearly identical with those of
the SOS editors on other DEC computer systems.

SOS   is   a   line-oriented editor;   that   is,   it   'sees'
text in terms of lines. A line is a string of characters
and spaces. Every line in an SOS file has a   line   number
associated   with   it.   When   SOS   displays   a line, it is
preceeded by its line number:

00100 This is a sample line of text.

SOS responds to the EDIT command   by   displaying   its
version   number and the full file specification. SOS then
prompts you to begin entering lines:

    @edit test.b16

    %File not found, Creating New file
    Input: TEST.B16.1
    00100

The line number prompt, '00100', indicates that you are in
INPUT MODE.   In this mode, each line you enter is placed
into the file.

Terminate   each   line   by   hitting   RETURN.   SOS
automatically   increments   the   line   number   by   100   and
prompts you for more input.

When you are finished inputting lines, hit the ESCAPE
(<esc>) key to get out of INPUT MODE. (On some terminals,
the equivalent key   is   labeled   'ALTmode',   'SELect',   or
'PREfix'.)   The   <esc> key is echoed to your terminal as a

dollar sign ($). Upon the echo of the $, you are in the
EDIT MODE. In this mode, SOS interprets each line you
enter as one of its special commands. The EDIT MODE
prompt is a '*'.

When you are done with the file, you can exit the
EDITor by entering an 'E' at the EDIT MODE level. SOS
will write the entered lines into a disc file under the
specified name and return you to TOPS-20 command level.

Summary of Frequently Used SOS Commands:

Command	Arguments*	Function
\<RET>	None	Print the next line in the file
\<ESC>	None	Print the previous line
P	position or range	Display line(s)
I	position	Insert new line(s) into file
N	increment and/or range	Renumber the lines in file
R	position or range	Replace one or more lines with new line(s)
D	position or range	Delete line(s) from the file
F	string\<ESC>	Find and print the next line containing the specified string

*key:

position	means you can specify a single line number
range	means you can specify a range of line numbers of the form \<firstline>:\<lastline>
increment	is a numeric value for line number incrementing
string	is any string of characters

For more information on the SOS Text EDITor, see:
DECSYSTEM-20 EDIT User's Guide

STEP 3: Compile the program.

The BLISS-16 source file that you've created must now be
compiled.   The BLISS-16c compiler package checks your
source program for syntax and programming errors, then
translates this input source file into a form suitable for
passing to an assembler. That code is then translated by
a cross-assembler (MACY11) into 'object' code, and placed
in a file called the 'object module file', which has the
file type, 'OBJ'.

You can invoke the BLISS-16 compiler by typing simply
'BLS16C'. The BLISS-16 compiler package will next display
its prompt:

    *

You should now enter the command line, which is of the
form:

    *objfil,lstfil=srcfill,srcfil2,.../switch/switch...

where either "objfil" or ",lstfil" (or both) may be
omitted and where all switches and all but the first
"srcfil" specification may be omitted.

Defaults for the relevant file extensions are:

    srcfil: B16
    lstfil: P11 (BLISS-11 listing file)
          LST (MACY11 listing file)
    objfil: OBJ
    intfil: I16 (BLISS-16c Intermediate file)

The BLISS-16 pre-processor translates the BLISS-16
source into BLISS-11 source, passing on the objfil and
lstfil specifications to the BLISS-11 compiler.

If both files are specified, MACY11 will be invoked
after BLISS-11 completes its compilation (assuming the
compilation was successful!) Objfil and lstfil
specifications from the BLISS-16c command line will be
passed on to MACY11 and will become the names of the
object and listing files output by MACY11. Both object
and listing files are always produced by MACY11 when
invoked from BLISS-16c.

If either objfil or lstfil is specified on the BLISS-16c

command line without the other, the one specified is taken
to be lstfil. In such case, MACY11 is not invoked, and
the assembly can then be performed either on the
DECSYSTEM-10/20 by MACY11, or on the PDP-11, itself.

The BLISS-16 pre-processor normally produces a BLISS-11
source file, called the intermediate file, or intfil, with
the file name taken from srcfil. This intermediate file
will normally be deleted by BLISS-11 when it is finished.
If MACY11 is invoked, the BLISS-11 listing file is also
normally deleted (after the assembly completes). Both
files (intfil and lstfil) can be optionally retained.

A number of switches can be specified on the command line.
(For a full list of switches, see Appendix A.)

STEP 4:   Link (Taskbuild) the object module.

The OBJECT module produced by the combined BLISS-16
pre-processor/BLISS-11 compiler/MACY11 cross-assembler
package is not in itself, executable: generally, an
object module contains references to other programs or
routines that must be bound with the object module before
it can be executed. This operation is the function of the
LINKER (or TASKBUILDER).

At this point (with the object file "in your hands") your
options are twofold:

- invoke one of the several PDP-11 simulator packages to
  taskbuild an executable image of your program, or
- transport the object file (from MACY11) to a PDP-11
  then taskbuild and run it there.

AN EXAMPLE OF THE WHOLE PROCEDURE:

Suppose that you wanted a BLISS program that looked like this:

```
MODULE SKELETON (MAIN = MAINLOOP) =
BEGIN
!++
!    This is a test BLISS program
!--
ROUTINE MAINLOOP: NOVALUE =
    BEGIN
    END;
END
ELUDOM
```

Let's look at the steps necessary to take this program from paper to an object file on the DECSYSTEM-20.

STEP 1: Log on to the system.

Make sure the terminal is powered ON. Hit CTRL-C. Let's say that USER-NAME, PASSWORD, and ACCOUNT are 'PEGRAM', 'PAL', and '432', respectively. In response to the opening message of TOPS-20 enter this line:

    @login pegram pal 432

Note that TOPS-20 will not echo the characters of the password (for the sake of security.)

The system will respond with a message like this:

    Job 15 on TTY52 14-Aug-78 00:34:22
    @

Note the system prompt (@) ending the message.

STEP 2:   Enter the BLISS-16 source program into a file.

        We will call the program 'NOFLAT.B16'.
        Here's the way the editing session would go.

        @edit noflat.b16
        %File not found, Creating New file
        Input: NOFLAT.B16.1
        00100   MODULE SKELETON (MAIN = MAINLOOP) =
        00200   BEGIN
        00300   !++
        00400   !   This is a BLISS test program.
        00500   !--
        00600   ROUTINE MAINLOOP: NOVALUE =
        00700   BEGIN
        00800   END;
        00900   END
        01000   ELUDOM$

The $ in the last line is the echoed <ESC> that you hit to
go from INPUT MODE to the EDIT MODE. SOS responds with

    *

The file looks OK, so exit SOS by entering 'E':

*E
[NOFLAT.B16.1]
@

STEP 3:   Compile the program.

Now you are ready to invoke the BLISS-16c translator package.   Let's say, as options, that you want a listing of the translation, statistics to be displayed, and to save all intermediate files created by the translator package.   The command strings to enter are:

```
@bls16c
*noflat,noflat=noflat/stat/noidelete
```

The BLISS-16c translator package will pre-process, compile, and assemble NOFLAT.B16.   If the translation is successful, the produced files will be:

- NOFLAT.I16 - the BLISS-11 equivalent of the original BLISS-16 source file.   (It is normally deleted.)

- NOFLAT.P11 - produced by the BLISS-11 compiler.   It contains the assembly language translation of the Intermediate BLISS-11 source file.)   It is also normally deleted.)

- NOFLAT.LST - the listing file produced by MACY11.

- NOFLAT.OBJ - the purpose of the whole exercise: the object file, ready for taskbuilding.   It's produced by MACY11.

STEP 4:   Link (Taskbuild) the object module.

You are now ready to taskbuild the object file, NOFLAT.OBJ,   and run it on either a simulator or a real PDP-11.

APPENDIX A

BLISS-16C Compiler Switches

The allowable command switches are:

Switch	Default	Description
/ADDRESS:opt	/ADDRESS:RELATIVE	Controls addressing mode to be used in the generated code.

- ABSOLUTE will cause an .ENABL AMA control directive to be emitted into the output file and addressing mode 37 (absolute) will be used instead of mode 67 (relative).

- RELATIVE will cause addressing mode 67 (relative) to be used.

Switch	Default	Description
/BLISS11:"switch(es)"	no switches	The switches specified within the quotes will be passed by BLISS-16C into the BLISS-11 command line. Example: BLISS11:"/S/A" will print routine lengths on the terminal as they are compiled and will allow EIS instructions.
/CODE	/NOCODE	Generate BLISS-11 code as a result of translation. /NOCODE is used to do a syntax check only.
/COMPRESS	/NOCOMPRESS	Compress the intermediate file. COMPRESS leaves out generated comments and indenting and replaces "BEGIN"-"END" with "("-")".

/DEBUG	/NODEBUG	Sets up an appropriate linkage for DEB16C (formerly called SIX12). The /DEBUG switch is passed to BLISS-11. However, the SIX12 debugger for BLISS-11 (developed at CMU) is not supported by this release.
/ERRS	/ERRS	Print warnings and error messages on the terminal.
/IDELETE	/IDELETE	Delete the intermediate file after BLISS-11 compilation. If MACY11 is invoked, delete the BLISS-11 listing file after the assembly completes.
/INTER:file	srcfil.I16	Override the default intermediate file name assignment with 'file'.

/LIST:(sw,...) or /LIST:sw	/LIST(COMMENTARY, NOEXPAND, NOREQUIRE, NOTRACE, OBJECT, SOURCE, SYMBOLIC)	COMMENTARY	List the BLISS-11 code produced by the translator
		EXPAND	Show result of macro expansion.
		OBJECT	Allow object code to be listed as indicated below. With NOOBJECT set no object code is listed.
		REQUIRE	List REQUIRE files.
		SOURCE	List BLISS-16C source code.
		SYMBOLIC	List the MACRO code generated by BLISS-11.
		TRACE	Trace macro

expansions.

/MACY11="switch(es)" no switches   The switches specified within the squotes will be passed by BLISS-16C via BLISS-11 into the MACY11 command line. A comma and/or an equals sign may occur within the quotes to show which switches are to be associated with which command line files. If no comma or equals sign is present, the switches are appended to the end of the MACY11 command line. Example:
MACY11=",/SP/NL=/EN:PNC" would associate /SP and /NL with the listing file and /EN:PNC with the source file.

/OBJECT:opt   /OBJECT:RELOCATABLE Controls object file format:

. ABSOLUTE will cause an .ENABL ABS control directive to be emitted into the output file. PSECT and EXTERNAL declarations will not be allowed.

. RELOCATABLE will generate an object file which must be processed by the task builder.

/OPTIMIZE   /OPTIMIZE   Perform optimization according to OPTLEVEL.

/OPTLEVEL:n   /OPTLEVEL:2   Controls amount of optimization:

0 = None
1 = Final only
2 = Code motion and Final

OPTLEVEL:0 is equivalent to

NOOPTIMIZE.

/PAGSIZE:n	/PAGSIZE:52	Set page size to "n" lines per page for the listing file. "n" is interpreted as a decimal number and must be between 20 and 52 inclusive. The default is 52.
/RBLISS11	/RBLISS11	Run the BLISS-11 compiler on the intermediate file.
/SAFE	/NOSAFE	Asserts, of the source code, that named variables will be addressed only by name. /NOSAFE is used when, because of the use of pointers to named variables, the value of a variable may be referenced or altered without the specific use of its name.
/STATISTICS	/NOSTATISTICS	Print routine names and sizes on terminal while compiling.
/UNAMES	/NOUNAMES	Generate unique names for OWN variables and non-global ROUTINE names when producing a listing which is to be assembled.
/VARIANT	/VARIANT:0	Set value of the %VARIANT predefined literal. If /VARIANT is specified without "n", the value is one.
/ZIP	/NOZIP	Optimize time at the expense of space.
/NO...		The prefix "NO" in a switch, as in /NOERRS generally reverses the sense of the switch. Exceptions to this are noted in the definitions above.

This page intentionally left blank.