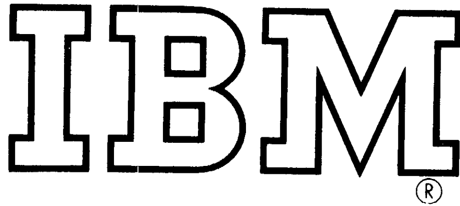
Z28-6642-0

BSL



BSL Language Specification

This publication is a complete specification of BSL, the Basic Systems Language. Topics covered include:

- Basic Structure
- Data Representation
- Data Manipulation
- Compile-Time Facilities

IBM Confidential

This document contains information of a proprietary nature. ALL INFORMATION CONTAINED HEREIN SHALL BE KEPT IN CONFIDENCE. None of this information shall be divulged to persons other than IBM employees authorized by the nature of their duties to receive such information or individuals or organizations authorized by the Systems Development Division in accordance with existing policy regarding release of company information.

PREFACE

All of the features of the BSL language are described in this publication. Other BSL publications are indicated below:

- Basic Systems Language Primer, Form Z28-6678, is intended as an introduction to BSL.
- BSL User's Guide, Form Z28-6682, contains implementation information for BSL.
- BSL Library (no form number) describes a set of procedures available to the BSL user when running his program.
- BSL Bulletin is an internally distributed newsletter covering various BSL topics.

Some features described in this publication are not available in early releases of the BSL compiler. These features are given below with the compiler release in which they are available.

- GENERATED Attribute - BSL/10
- Compile-Time Facilities - BSL/10
- OPTIONS(VLIST) Attribute - BSL/11

This publication is a revision of the BSL Basic Systems Language Description, dated January 24, 1968.

Copies of BSL publications are available from Department D76, Systems Development Division, Poughkeepsie, New York. Requests for having names added to the BSL Bulletin Distribution List should also be directed to this address.

CONTENTS

INTRODUCTION 7

BASIC STRUCTURE 8
Syntactic Structure 8
    Character Set 8
    Identifiers 9
    Blanks 9
    Comments 9
    Source Input 10
Program Structure 11
    Statements 11
    Groups 12
    Procedures 13

DATA REPRESENTATION 14
Declarations 14
Data Types 15
    Arithmetic Items 15
    String Items 15
    Pointer Items 16
    Program Items 17
Organization 18
    Arrays 18
    Structures 18
    Arrays of Structures 20
Scope 20
Storage Class 22
    Fixed Data Areas 22
    User Generated Data 23
    Automatic Storage Allocation 23
    Data in Registers 24
    Parameters 24
    Indirect Addressing 24
Other Attributes 26
    Boundary 26
    Initialization 27
    Normality 28
Default Attributes 29
    Implicit Declaration 29
    Default Data Type 29
    Default Precision and Length 29
    Default Scope 29
    Default Storage Class 30
    Default Boundary 30
    Default Initialization 30
    Default Normality 30

CONTENTS (continued)

DATA MANIPULATION 31
Value Assignment 31
Expressions 32
    Operators 32
    Associating Operators and Operands 32
    Comparison Operators 32
Arithmetic Operations 33
    Mixed Precisions 33
    Assignment Involving Mixed Precisions 33
    Arithmetic Operations with String Items 33
String Operations 34
    Operations with Unequal Lengths 34
    Assignment Involving Unequal Lengths 34
    String Comparisons 34
    String Operations with Arithmetic Items 34
Mixed Types 35
    Subscripts and Substrings 35
    Assignments 35
    Comparisons 35
    DO Terms 35
    Argument Expressions 35
Statements 36
    The Assignment Statement 36
    The CALL Statement 37
    The DECLARE Statement 38
    The DO Statement 38
    The END Statement 39
    The ENTRY Statement 39
    The GENERATE Statement 40
    The GOTO Statement 41
    The IF Statement 41
    The Null Statement 42
    The PROCEDURE Statement 42
    The RELEASE Statement 43
    The RESTRICT Statement 44
    The RETURN Statement 44

COMPILE TIME FACILITIES 45
Basic Structure 45
    Macro Statements 45
    Macro Variables 46
    Source Text Replacement 46
    Rescanning 47
Macro Statements 48
    Macro Declaration 48
    Value Assignment 49
    Scan Control 50
    Text Inclusion 51
    Conditional Execution 51
    Macro Activation 52

CONTENTS (continued)

ADDITIONAL TOPICS 54
Substring Notation 54
    Variable Length Substrings 55
    Bit Substrings 55
Builtin Functions 56
    ABS Builtin Function 56
    ADDR Builtin Function 56
Pointer Association 57
Register Usage 58
Procedure Options 59
    The CODEREG Option 59
    The DATAREG Option 59
    The REENTRANT Option 60
    The SAVE Option 60
    The DONTSAVE Option 60
    The NOSAVEAREA Option 61
    Combining Options 61
Name Placeholder 62
Variable Parameter Lists 63

APPENDIX I: LANGUAGE KEYWORDS 65

APPENDIX II: ATTRIBUTE CONFLICTS 67

INDEX 68

INTRODUCTION

A systems programming tool must allow access to the basic registers and functions of a machine, and must be able to satisfy constraints on program size, speed, and environment. These requirements can be met by assembly language, since one can use it to code at the machine instruction level. However, much of this coding effort is directed at satisfying assembler and machine requirements not related to the problem being solved.

BSL, the Basic Systems Language, allows the programmer to concentrate on the solution of the problem without being distracted and slowed by non-essential considerations. BSL provides access to the machine when required by the problem. Where not required, such machine detail can be avoided and a more convenient language level used.

BASIC STRUCTURE

BSL allows the user to write his program in a free field format. Input program text is examined as a single string of characters, continuing from one input record to the next. Program elements are delimited by special symbols rather than by record boundaries or positions.

SYNTACTIC STRUCTURE

Character Set

The character set used for BSL is EBCDIC. Character-type data appearing in the BSL program is interpreted using EBCDIC character codes.

In writing a BSL program, all 256 EBCDIC character codes may be used within comments and character data constants. The rest of the program is written using only a subset of EBCDIC consisting of the letters A through Z, the digits 0 through 9, and the special characters given below:

<table>
<tr><th>Name</th><th>Graphic</th></tr>
<tr><td>Blank</td><td></td></tr>
<tr><td>Equal or assignment symbol</td><td>=</td></tr>
<tr><td>Plus or addition symbol</td><td>+</td></tr>
<tr><td>Minus or subtraction symbol</td><td>-</td></tr>
<tr><td>Asterisk or multiplication symbol</td><td>*</td></tr>
<tr><td>Slash or division symbol</td><td>/</td></tr>
<tr><td>Left parenthesis</td><td>(</td></tr>
<tr><td>Right parenthesis</td><td>)</td></tr>
<tr><td>Comma</td><td>,</td></tr>
<tr><td>Semicolon</td><td>;</td></tr>
<tr><td>Colon</td><td>:</td></tr>
<tr><td>Quote mark</td><td>'</td></tr>
<tr><td>Negation symbol</td><td>¬</td></tr>
<tr><td>And symbol</td><td>&</td></tr>
<tr><td>Or symbol</td><td>|</td></tr>
<tr><td>Greater than symbol</td><td>></td></tr>
<tr><td>Less than symbol</td><td><</td></tr>
<tr><td>Percent or compile-time symbol</td><td>%</td></tr>
</table>

Letters and digits are used in forming language keywords, data constants, and the user's data names. The special characters are used, either singly or in combination, to form operators and other delimiters in the language. (The term 'composite delimiter' applies to those operators and delimiters composed of more than one character.)

Identifiers

An identifier is a string of letters and digits, with the first character alphabetic. Identifiers are used as names of data items and as language keywords. Those identifiers used as data names must be eight or less characters in length. They should not conflict with keywords, since some keywords are reserved for the implementation. A list of BSL keywords is given as an appendix to this manual.

Examples of identifiers:

```
IDENT          legal; may be used as a data name
NUM999         legal; may be used as a data name
MORETHAN8      illegal; more than 8 characters
2BITS          illegal; first character is not a letter
$35            illegal; first character is invalid
CALL           BSL keyword; illegal as a data name
```

Blanks

Blanks may appear in the program text, with certain minor requirements on their position. They may not be embedded within identifiers (including keywords), data constants, or composite delimiters. Identifiers or constants may not be immediately adjacent; they must be separated by some appropriate delimiter (which could be a blank). Blanks may otherwise be freely introduced as desired.

Examples:

```
AA=BB+CC; is equivalent to AA = BB  +  CC   ;
  but not equivalent to AA=B  B+CC;

DOI=JBYKTOL; is equivalent to DOI  = JBYKTOL     ;
  which is the same assignment statement,
  but not equivalent to DO I=J BY K TO L;
  which is an entirely different DO statement
```

Comments

Comments may be included within the program text. A comment has the form:

```
/* [data-character-string] */
```

A comment may be introduced wherever a blank is permitted. The character string in the comment may not include the substring '*/', since this would be recognized as terminating the comment.

Examples:

```
/*THIS IS A COMMENT*/
/**/
/*THE STRING '/*' IS OK     */
/*THE STRING '*/' IS NOT OK */  incorrect comment
```

<u>Source Input</u>

BSL source statements are usually written in columns 1 through 72 of an input record, with columns 73 through 80 available to the user for information such as sequencing. Other margins may be specified, as discussed in the BSL User's Guide. The term 'input line' refers to that part of the input record used for writing BSL source text.

Record boundaries and positions are not usually significant in writing BSL source statements. One exception will be found with the GENERATE statement, discussed in the section on that statement. Another is where BSL symbols conflict with system usage (such as /* in columns 1 and 2 for Operating System/360); these conflicts are discussed in the BSL User's Guide.

Identifiers, arithmetic constants, and composite delimiters may not be split across input lines. Comments and string constants may be spread across a number of input lines.

Examples, with input lines enclosed by { and }:

```
{    A=0; B=0; /*multiple statements on one line*/   }
{    DECLARE   /*spread across a number of lines*/   }
{      STRING CHARACTER(100),                        }
{      FLAGS BIT(8);                                 }
{                                                    }
{              /*The following is incorrect*/ THIS=NO}
{GOOD;         /*NOGOOD can not be split*/           }
{                                                    }
{              /*A comment may span                  }
{                input lines*/                       }
```

PROGRAM STRUCTURE

The solution of a programming problem is specified as one or more BSL procedures. A procedure is a sequence of basic elements, called statements, expressing the data and operations for the procedure. The procedure might express the total problem solution or it might express a self-contained segment, separated because it is frequently used or because it is a convenient division of coding effort. A procedure is called upon by other procedures whenever they have a need for its particular function.

Statements

Statements are the basic structural units of a procedure. BSL statements have the form:

```
[identifier:]...  statement-keyword  statement-body  ;
```

The assignment statement, the null statement and the IF statement are slight variations of this form.

The leading identifier is the label or entry name associated with the statement. Entry names (which must precede PROCEDURE and ENTRY statements) indicate entry points into a procedure. Labels (which optionally precede other statements) indicate transfer points within a procedure. More than one label can appear on a statement used as a transfer point, but only one entry name can appear on a statement which is an entry point.

The statement keyword is an identifier characterizing the type of statement. The statement body will vary for different statement types.

Examples:

```
     GOTO LABEL3;          /*a GOTO statement*/
     CALL P(A,B);          /*a CALL statement*/
P:   PROCEDURE(X,Y);       /*a PROCEDURE statement, with
                               entry name P*/
LL:  A=B+C-D;              /*an assignment statement, with
                               label LL. Note that there is
                               no statement keyword*/
     IF A>B THEN           /*an IF statement, which     */
       GOTO LL;            /*  includes other statements*/
NO:  ;                     /*a null statement, with label
                               NO. Note that there is no
                               statement keyword and no
                               statement body*/
```

Groups

Several statements can be grouped together so that their execution is controlled as a unit. This facility is especially useful for conditional and iterative execution of more than one statement. Such collections of statements are known as groups. Groups have the form:

```
DO-statement

[group-member]...

END-statement
```

The DO statement heads a group, and may specify iterative execution of the enclosed statements. The end of a group definition is indicated by the END statement, which may refer back to the label (if any) on the starting DO statement. A member of a group may be a single statement or it may be another, nested group.

Examples:

```
G1:  DO;                     /*start of a simple group*/
     X=0;

G2:  DO I=1 TO 10;           /*start of an iterative,
                               nested group*/
     A(I)=B(I);
     END G2;                 /*end of the iterative group*/

     IF X=0 THEN             /*conditional execution*/
          DO;                /*start of a conditionally
                               executed group*/

          CALL P;
          X=1;
          GOTO G2;
          END;               /*end of the conditionally
                               executed group*/
     END G1;                 /*end of the simple group*/
```

Procedures

A procedure is a sequence of statements defining a self-contained section of a program. Procedures have the form:

PROCEDURE-statement

[procedure-member]...

END-statement

The PROCEDURE statement heads a procedure, specifying the name for that entry point as well as any parameter or option information for that procedure. Secondary entry points can be defined with the ENTRY statement. The end of a procedure definition is indicated by the END statement, which may refer back to the name on the PROCEDURE statement.

A member of a procedure may be a single statement, a statement group, or another, nested procedure. A nested procedure is local to its immediately encompassing procedure, and may only be invoked within that procedure. It can work with the data of encompassing procedures, and in addition can have its own data.

Examples:

```
MP:  PROCEDURE;             /*start of procedure MP*/

LOOP:SWITCH=0;
     CALL PDIV;             /*PDIV is a nested procedure*/
     IF SWITCH¬=0 THEN GOTO LOOP;

     RETURN;                /*return from MP to the
                              procedure calling it*/
PDIV:PROCEDURE;             /*start of PDIV, within MP*/
     IF V>0 THEN
          DO;               /*start of a group*/
          SWITCH=1;
          V=V-1;
          END;              /*end of the group*/

     RETURN;                /*return from PDIV to MP*/
     END;                   /*end of procedure PDIV*/

     END MP;                /*end of procedure MP*/
```

DATA REPRESENTATION

A data item in BSL has a set of attributes indicating such things as its type, location, and organization. These attributes are associated with the name representing the data item in the procedure.

DECLARATIONS

The user explicitly indicates the attributes associated with a name by means of the DECLARE statement. This statement has the basic form:

```
{DCL    }
{DECLARE} name [attribute]... [, name [attribute]... ]... ;
```

More than one DECLARE statement can be used in a procedure, but a particular name may be declared only once. All data names do not have to be declared; a set of default attributes will apply for undeclared items, based on their use within the procedure. Defaults will also be applied to declared items as necessary to complete their attribute set.

Any declaration of a data item must appear before a use of that item. An exception exists in the case of parameters in a PROCEDURE statement, which may be declared after their use in that statement.

Attributes common to several declared names can be factored to eliminate repeated specification. This is achieved by enclosing the unique parts of the declarations for the items in parentheses, and following this by the common attributes. The partial declarations may themselves involve factoring. Factored attributes must not conflict with attributes within the parentheses.

Examples:

```
DECLARE  A FIXED,  B  BIT(8);
DECLARE  (FLAGA,FLAGB) BIT(1);
DCL  (B BIT(8) , C CHAR(256) ) EXTERNAL, V FIXED;
  /*B is declared as BIT(8) and EXTERNAL*/
  /*C is declared as CHAR(256) and EXTERNAL*/
  /*V is declared as FIXED*/
DCL  (LV FIXED,(SB BIT(1),SC CHAR(16))EXT)LOCAL;
  /*LV is declared as FIXED and LOCAL*/
  /*SB is declared as BIT(1), EXT and LOCAL*/
  /*SC is declared as CHAR(16), EXT and LOCAL*/
```

DATA TYPES

Data items in BSL can be classified as arithmetic, string, pointer, or program types.

<u>Arithmetic Items</u>

Attributes: FIXED [(precision)]

Arithmetic data is carried internally as a binary fixed point integer. For convenience, arithmetic constants in source text may be in either binary or decimal form. Such a decimal constant consists of one or more adjacent decimal digits. A binary constant consists of one or more binary digits (0 or 1) followed by the letter B.

'Precision' is the number of bits assigned for the maximum positive value of the item. An additional bit is assigned as a sign bit; the total field represents a value in two's complement form. Precisions of 15 and 31 are allowed, corresponding to halfword and fullword quantities. Absence of a specified precision results in a default precision of 31.

```
     Examples of declarations:
          DECLARE VAR FIXED;
          DECLARE HALFWORD FIXED(15);
     Examples of source text constants:
               743      decimal constant
               1011     decimal constant
               1011B    binary constant; decimal value=11
               114B     invalid binary constant
               40FF     invalid constant
```

<u>String Items</u>

```
     Attributes:     BIT    (length)

                    {CHAR     }
                    {CHARACTER}  (length)
```

String data is carried internally as either sequences of bits or sequences of characters (i.e., bytes). For convenience, string constants in source text may be in character, bit, or hexadecimal form. A character string constant consists of one or more EBCDIC-coded characters enclosed by quote marks, with enclosed quote marks represented by two adjacent quote marks. A bit string constant consists of one or more binary digits (0 or 1) enclosed in quote marks and followed by the letter B. A hexadecimal string constant consists of one or more hexadecimal digits (0 through 9 and A through F) enclosed in quote marks and followed by the letter X.

Examples of declarations:

```
DECLARE BVAR BIT(16);
DECLARE CFOR1 CHAR(1), CFOR256 CHARACTER(256);
```

Examples of source text constants:

```
'ABC'          character constant
'IT''S'        character constant
'10'           character constant
'10'B          bit constant
'10'X          hex constant; bit value='00010000'B
'F3'X          hex constant; bit value='11110011'B
'13'B          invalid bit constant
'G0'X          invalid hex constant
```

Pointer Items

Attributes: { PTR | POINTER } [(precision)]

Pointers are used to address data indirectly. The pointer serves to locate a data area whose attributes are described by a BASED data item, explained in the section on Storage Class. Pointers have values which are positive integers; they may be used as arithmetic items.

'Precision' is the number of bits assigned for the maximum value of the pointer. Precisions of 8, 16, 24, 32, 15, and 31 are provided. Pointers of precisions 8, 16, 24, and 32 are assigned 1, 2, 3, and 4 bytes respectively, in which all bits are significant. Pointers of precisions 15 and 31 are assigned 2 and 4 bytes respectively, which includes a leading (sign) bit that will always be 0. The difference between precisions 15 and 16 (and 31 and 32) is that the generated code doesn't have to guard against propagating the left bit for precision 15, as would be done in an AH instruction for example. Absence of a specified precision results in a default precision of 31.

Examples:

```
DECLARE P POINTER;
DECLARE PTOV PTR(24),PLOCORE PTR(15);
```

Program Items

Attributes: LABEL

ENTRY

This type encompasses the statement labels and entry names of a program. ENTRY applies to entry names appearing on ENTRY and PROCEDURE statements in the defining procedure, and also applies to entry points which are in other procedures but are known to the declaring procedure. LABEL applies to labels on statements in a procedure, and also applies to transfer points which are in other procedures but are known to the declaring procedure. Labels and entry names do not have to be declared; they can receive an implicit declaration, as explained in the section on Default Attributes.

Indirect program addressing can be accomplished by using labels or entries with the BASED attribute. This is discussed in the section on Storage Class.

Examples:

```
A:     PROCEDURE;      /*A is an ENTRY for this procedure*/
       DECLARE
         P ENTRY,      /*P is an ENTRY for some separately
                         defined procedure*/
         LIN LABEL,    /*LIN is a LABEL appearing later
                         in the procedure*/
         PT POINTER, /*PT will be used with INDIRECT for */
         INDIRECT LABEL BASED(PT); /*indirect addressing*/
     CALL P;
     GOTO COM;         /*COM is a LABEL appearing later in
                         the procedure*/
E:   ENTRY;            /*E is an ENTRY for this procedure*/
COM: PT=ADDR(LIN);     /*COM is the LABEL referenced in
                         the earlier GOTO statement*/
     GOTO INDIRECT; /*The transfer point is determined
                         from the value of the pointer PT*/
LIN: RETURN;           /*LIN is the LABEL indicated in
                         the earlier DECLARE statement*/
     END A;
```

ORGANIZATION

The data for a program may include single, unrelated data items; these items are known as scalar items. Data also may be organized either as a collection of identical data items or as an arrangement of possibly dissimilar data items. The corresponding forms in BSL are arrays and structures.

Arrays

An array is a collection of identical data items. The collection has a common name, called the array name. The array elements are organized consecutively in storage. A particular element is referenced by using the array name followed by an expression in parentheses whose value indicates the appropriate element. This referencing notation is known as subscript notation.

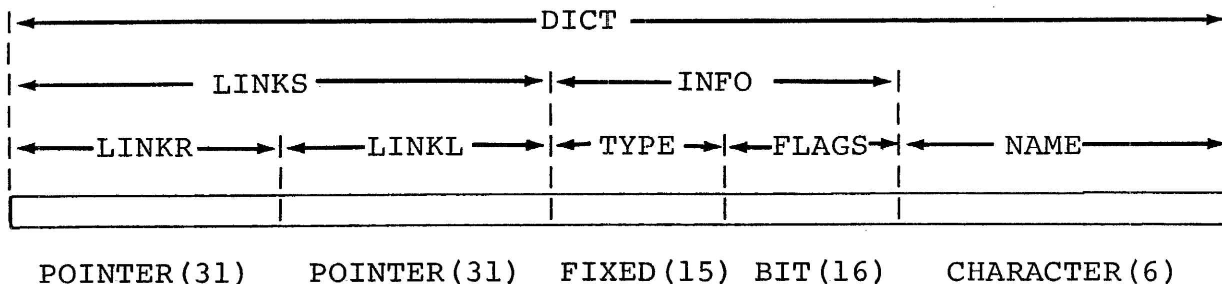
The number of elements in an array is declared in a dimension attribute. This attribute consists of a decimal number in parentheses immediately following the array name.

Examples:

```
DECLARE ARRAY(10);            /*ARRAY has ten elements*/
DECLARE C2(32) CHAR(2);       /*C2 is an array of 32 two-character
                                elements*/
ARRAY(6) = 0 ;                /*The sixth element of ARRAY is set
                                to zero*/
C2(I) = '  ' ;                /*The Ith element of C2 is set to
                                blanks*/
```

Structures

A structure is an arrangement of scalar items, arrays, and other structures. The structure has a name, called the structure name, and each component of the structure will have its own name. Reference to a component uses the component's name. Consider the following organization:



The overall structure is called DICT. Its initial component is the structure LINKS, which has as its components the pointers LINKR and LINKL. The next component of DICT is the structure INFO, which consists of the arithmetic item TYPE and the bit string FLAGS. The final component of DICT is the scalar item NAME, which is a character string.

Structuring is indicated in BSL by level numbers preceding the structure and component names in a declaration. The outermost structure must have a level number of 1. Components of a structure must have a level number greater than that of the structure; this number should be the same for all components of the structure. The following declaration corresponds to the sample structure:

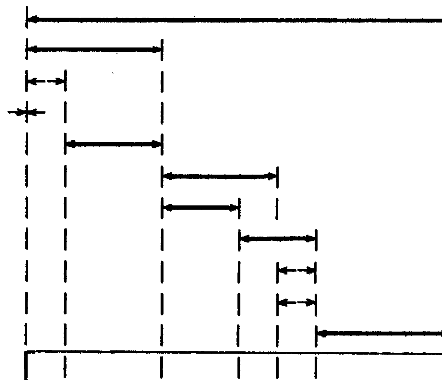
```
DECLARE /*SAMPLE STRUCTURE*/
   1 DICT,
        2 LINKS,
              3 LINKR POINTER(31),
              3 LINKL POINTER(31),
        2 INFO,
              3 TYPE FIXED(15),
              3 FLAGS BIT(16),
        2 NAME CHARACTER(6);
```

A structure name represents a data item, distinct from the structure's components. It has a set of data attributes, either explicitly declared or applied as a default. The default applied is CHARACTER, with a length sufficient to span all components.

A structure may be declared with a size different from the size required to span its components. Where the structure size is smaller than the required size, the structure's components may overlap the components following in an outer structure. Knowledge of how data is kept internally is necessary for using this facility.

Example, with mapping:

```
DECLARE
  1 SPECIAL,
    2 FIELD POINTER(31),
      3 SWITCHES BIT(8),
        4 FIRSTSW BIT(1),
      3 P24 POINTER(24),
    2 TOOSMALL CHAR(3),
      3 VAR FIXED(15),
      3 OVER CHAR(2),
    2 BITOVER BIT(8),
      3 CHAROVER CHAR(1),
    2 LAST FIXED(31);
```

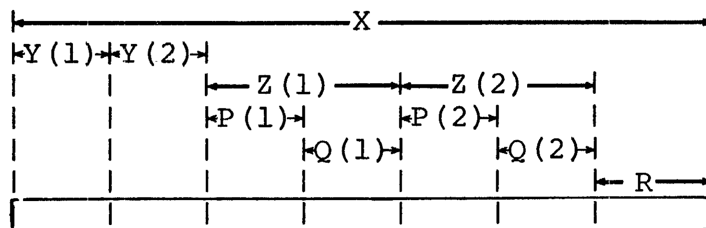


Arrays of Structures

An array of structures is indicated by specifying a dimension attribute for a structure name. This dimension attribute applies to all components as well as to the structure name. References to such a structure or its components use subscript notation as for any other array. The components of a structure array may not themselves have a dimension attribute.

Example, with mapping:

```
DECLARE
     1 X,
       2 Y(2),
       2 Z(2),
         3 P,
         3 Q,
       2 R;
```



SCOPE

Attributes: { INT | INTERNAL }

{ EXT | EXTERNAL }

The scope of a name is that portion of the program where the data item represented by the name is known and can be referenced. A data item is known in a set of separately compiled procedures if each procedure declares the name for that item as EXTERNAL; the name represents the same data item in each procedure. A data item is known only within a single procedure if that procedure declares the name as INTERNAL; the same name declared as INTERNAL in different procedures represents different data items in each procedure.

An entry name for a procedure not internal to any other procedure has EXTERNAL scope; such a procedure is called an external procedure. Entry names for procedures defined within other procedures are INTERNAL, local to the containing procedure. Statement labels are INTERNAL, local to the procedure containing the statement.

Names known in a procedure are also known in procedures internal to that procedure. (The internal procedures should not declare these names.) However, names of item internal to a procedure are not considered known in any containing procedure, and therefore cannot be referenced there.

Examples:

```
P:   PROCEDURE;             /*P is EXTERNAL*/
     DCL A  INTERNAL,
         EV EXTERNAL,
         (INT1,INT2) INTERNAL ENTRY;
     A=0;
     GOTO LAB;
EP:  ENTRY;                 /*EP is EXTERNAL*/
     A=1;
LAB: CALL INT1;             /*LAB is INTERNAL to P*/
     CALL INT2;
     EV=A;
     RETURN;
INT1:PROCEDURE;             /*INT1 is INTERNAL to P*/
     DCL INTLV;             /*INTLV is INTERNAL to INT1*/
     DO INTLV=1 TO 10;
     A=A+INTLV;             /*valid reference to A*/
     END;
     RETURN;
     END INT1;
INT2:PROCEDURE;             /*INT2 is INTERNAL to P*/
     INTLV=0;               /*invalid; INTLV is internal
                              to INT1, and is not known
                              in INT2*/
     CALL INT1;             /*valid; INT1 is known in P,
                              and therefore also in INT2*/
     RETURN;
     END INT2;
     END P;
```

```
EXTERNAL items:  P, EP, EV
Items INTERNAL to P:  A, LAB, INT1, INT2
Items INTERNAL to INT1:  INTLV
Items INTERNAL to INT2:  none

Items known externally:  P, EP, EV
Items known in P:  P, EP, EV, A, LAB, INT1, INT2
Items known in INT1:  P, EP, EV, A, LAB, INT1, INT2,
                      INTLV
Items known in INT2:  P, EP, EV, A, LAB, INT1, INT2
```

STORAGE CLASS

Data items may be classifed according to how they are located. An item may be fixed at a particular location, or it may have a position which varies depending on a locating mechanism.

```
Attributes:     STATIC
                     LOCAL
                     NONLOCAL

                GENERATED

                { AUTO      }
                { AUTOMATIC }

                { REG      }
                { REGISTER }   (register)

                 BASED  [ (locating-expression) ]
```

Fixed Data Areas

The STATIC attribute indicates that main storage is statically assigned for the data item, and never reassigned. STATIC has subclasses LOCAL and NONLOCAL, which indicate the location of the data item relative to the generated code for the declaring procedure.

The LOCAL attribute indicates that the data item is assigned storage, in the same area (CSECT) as for the generated code.

The NONLOCAL attribute specifies that the item is not assigned storage by the declaring procedure. In the case of a NONLOCAL EXTERNAL item, storage assignment is provided by a declaration as LOCAL EXTERNAL in some separately compiled procedure.

Examples:

```
P:PROCEDURE;
  DECLARE L INTERNAL LOCAL STATIC;
       /*storage for L is with the code for P*/
  DECLARE NL STATIC NONLOCAL;
       /*storage for NL is assigned by some other
         procedure having NL as LOCAL EXTERNAL*/
```

User Generated Data

The GENERATED attribute is associated with items defined and insured addressable by the user in a GENERATE statement (explained later on). These items are internal to the procedure, but are not assigned a storage area by the compiler.

The attribute combination NONLOCAL INTERNAL, formerly used to obtain this function, is still recognized by the compiler but should no longer be used.

Examples:

```
  DECLARE DCBISAM GENERATED CHAR(200);
    •
    •
    •
  GENERATE; /*definition of DCBISAM*/
DCBISAM DCB   •••
              •••
$ENDGEN
```

Automatic Storage Allocation

The AUTOMATIC attribute may be used in a reentrant environment to provide an automatic allocation of storage for data on entry to a procedure, and an automatic freeing on exit. (The REENTRANT option is discussed under Procedure Options.) AUTOMATIC should not be used in a nonreentrant environment.

AUTOMATIC data declared in internal procedures will be allocated at the same time as AUTOMATIC data for the outer procedure. This reflects the fact that internal procedures cannot be separately reentrant, and thus only require one data allocation for each allocation of the outer procedure.

Examples:

```
P:PROCEDURE OPTIONS (REENTRANT);
  DECLARE S CHAR(256) AUTOMATIC; /*storage assigned on
    •                              entering P*/
    •
    •
Q:PROCEDURE;
  DECLARE SINQ BIT(32) AUTOMATIC; /*storage also assigned on
    •                               entering P*/
    •
    •
  END Q;

  END P;
```

Data in Registers

The user may associate the REGISTER attribute with a name, to access data located in the registers of the machine. Register specification must be in the range 0 through 15, corresponding to the general registers on System/360. Use of registers requires knowledge of the conventions used by the compiler. These are discussed in the section on Register Usage.

Example:

```
DECLARE R3 REGISTER(3) POINTER(31);
```

Parameters

For some names, the data attributes provided in the declaration are applied to an area located indirectly. The most common example of this is the use of input parameters for a procedure. References to parameters are indirect references through a list of pointers to the corresponding arguments.

Parameters have no attribute keyword to represent 'parameter' storage class. Parameters are indicated as such by their appearance in a parameter list.

Examples:

```
P:PROCEDURE (PARAM1,PARAM2);  /*PARAM1 and PARAM2 are
                                parameters*/
  DECLARE PARAM1 FIXED(31), PARAM2 CHAR(16);
    •
    •
    •
  END P;
```

Indirect Addressing

The user may obtain indirect data addressing by using the BASED storage class. A name in this class provides a description of an area whose location is based on an associated pointer value. This pointer value is the value of the locating expression in the BASED attribute, or of a pointer associated at a reference to the BASED item. (This topic is explained in the section on Pointer Association.) The expression or pointer yields the address of a storage area. This area is then treated as if it had the attributes associated with the BASED name.

The locating expression is of the form:

```
 ( pointer               [ +decimal constant ] )
 { ADDR(data-item)       [ -decimal-constant ] }
 ( decimal-constant                            )
```

A name based on a decimal constant represents data starting at the absolute machine location indicated by the constant. A name based on the address function (explained under Builtin Functions) represents the same area as the item given as the function argument; the attributes do not have to be the same, and may describe the area completely differently. The description used for an area depends on the name referenced.

Examples:

```
DCL SVCOPSW BASED(32);
DCL V STATIC LOCAL;
DCL MAPSV BASED(ADDR(V));
      /*MAPSV represents the same area as V*/
DCL PART2V BASED(ADDR(V)+2) FIXED(15);
      /*PART2V represents the second half of V*/
DCL (P,PT) LOCAL POINTER;
DCL DCHAR CHAR(1) BASED(P);
      /*DCHAR describes an area located by P*/
DCL DARITH FIXED BASED(P);
      /*DARITH describes an area located by P*/
DCL DCBIT1 BIT(1) BASED(ADDR(DCHAR));
      /*DCBIT describes an area located by the
        address of DCHAR, being the value of P*/
DCL DLABEL LABEL BASED(PT);
      /*DLABEL describes a program transfer point
        located by PT.  GOTO DLABEL; would cause
        a transfer to the point indicated by the
        value of PT*/
```

OTHER ATTRIBUTES

Boundary

```
Attributes:  { BDY      }
             { BOUNDARY }  (boundary [,position] )
```

BOUNDARY provides a means of aligning data items on System/360 storage boundaries. The boundary may be BYTE, HWORD, WORD, or DWORD, corresponding to byte, halfword, word and double word boundaries, respectively. The position is a decimal integer selecting a byte position within the boundary field; if unspecified, it is assumed to be 1. Data items have default boundaries appropriate to their representation. Arrays have the same boundary for each element. Components of a structure must have a basic boundary less than or the same as that of the structure; the positions within that boundary need not agree.

Note that positions in lesser boundaries have more than one alternative with respect to the higher boundaries. WORD,1 when considered within a doubleword, for example, could be either DWORD,1 or DWORD,5.

Examples:

```
DECLARE P PTR(24) BOUNDARY(WORD,2);
DECLARE C CHAR(80) BOUNDARY(WORD);
DECLARE 1 A BOUNDARY(WORD),
          2 B(2) BDY(HWORD),
            3 C CHAR(1) BDY(HWORD,2), /*legal*/
          2 D BIT(8) BDY(DWORD,6); /*illegal,
            since DWORD is greater than WORD*/
```

Initialization

Attributes: {INIT | INITIAL} (value [,value]...)

Initial values may be specified only for STATIC LOCAL data items. The values are taken as the initial contents of the storage area assigned for the data item. No assumptions should be made as to the initial contents of uninitialized data items. A user should also be careful in using and depending on an initial value for a data item which may be modified.

Multiple initial values are used for array initialization. If not enough values are specified, the remaining array elements will not be initialized; if too many values are specified, the leftover ones are ignored. An asterisk may be used in place of an initial value to indicate no initialization for a particular element. For convenience, an initial value may be preceded by a replication number in parentheses, indicating repetition of that value the appropriate number of times.

The BSL compiler has some restrictions concerning initial values which do not match the item being initialized. These restrictions can be found in the BSL User's Guide.

Examples:

```
DCL V FIXED INITIAL(0);
DCL C CHAR(2) INIT('XX');
DCL AR1(5) FIXED INIT(1,2,*,4,5),
    AR2(5) FIXED INIT((5)0),
    AR3(5) FIXED INIT((2)*,(2)2,0);
DCL 1 FLAGS BIT(8) INIT('00'X),
      2 DEFFLG BIT(1),
      2 NMFLG BIT(1);
DCL P POINTER INIT(ADDR(V));
DCL X FIXED INIT('A');    /*restricted*/
DCL B BIT(1) INIT(1);     /*restricted*/
```

Normality

Attributes: NORMAL

{ABNORMAL | ABNL}

The need for NORMAL and ABNORMAL attributes arises from compiler optimization which may be incorrect for ABNORMAL data.

Consider two variables, A and B, which overlap in some unknown way. At a reference to A, the compiler may generate a register load, and may keep history on that register, remembering that it contains the value of A. After an assignment to B, this history would be incorrect, but the compiler would not know it was incorrect. The next reference to A would still make use of this history, and thus would have incorrect results. To avoid this, the user could declare A and B as ABNORMAL, restricting history about these variables and thus providing correct results.

ABNORMAL is not necessary for overlaps known to the compiler, as for example with a structure and its components. It is only necessary for overlaps unknown to the compiler, as might exist with EXTERNAL items, parameters, or BASED items.

Examples:

```
P:  PROCEDURE(A,B);  /*A and B represent the same
                       item, with value=1*/
    V=A;             /*V is set to the value of A,
                       which is 1*/
    B=2;             /*B (and thus A) is set to 2*/
    V=A;             /*V would be incorrectly set to
                       1, the old value of A, if history
                       were kept.  The statement DECLARE
                       (A,B) ABNORMAL is necessary to
                       assure no history*/
    DCL Q POINTER,
        D BASED(Q),
        C INIT(1);
    Q=ADDR(C);       /*C and D now overlap*/
    V=C;             /*V is set to the value of C,
                       which is 1*/
    D=2;             /*D (and thus C) is set to 2*/
    V=C;             /*V would be incorrectly set to
                       1, the old value of C, if history
                       were kept.  Declaring C and D
                       ABNORMAL is necessary to assure
                       no history*/
```

DEFAULT ATTRIBUTES

If the user chooses not to declare an item, or if the declaration does not provide all the attributes of an item, then defaults will apply as necessary to provide a complete attribute set. Defaults are applied at the first appearance of an item, whether this is as a label, as an entry name, in a declaration, or in a reference within the procedure. (Note that this would generally require a declaration of an INTERNAL ENTRY.) The exception is parameters in a PROCEDURE statement, which would have defaults applied at the next appearance.

Implicit Declaration

Labels and entry names may receive implicit declarations by their use within a procedure. If the first reference is as the target in a CALL statement or as the name preceding a PROCEDURE or ENTRY statement, then that identifier is implicitly declared as ENTRY. If the first reference to an identifier is as the target in a GOTO statement or as the name preceding a statement other than PROCEDURE or ENTRY, then that identifier is implicitly declared as LABEL.

Items appearing in the BASED expression or in a pointer association position (discussed under Pointer Association) do not receive implicit declaration as pointers. Pointers must be explicitly declared as such, and must be declared before their appearance in pointer contexts.

Default Data Type

For scalar items and array items the default type is FIXED. For structures, the default is CHARACTER.

Default Precision and Length

The default precision for both FIXED and POINTER items is 31 (equivalent to a full word). Default length for structures defaulted to CHARACTER is whatever is sufficient to span the components. BIT and CHARACTER items both require explicit length specification.

Default Scope

Entry items, other than those which appear as entry names for internal procedures, default to EXTERNAL. Other than these cases, the default scope is INTERNAL.

Default Storage Class

Entry point parameters are in a 'parameter' storage class, for which no attribute keyword is defined.

Items declared as LOCAL or NONLOCAL imply STATIC.

Items declared as STATIC will default to NONLOCAL if they are EXTERNAL and not initialized, and will default to LOCAL if they are INTERNAL or both EXTERNAL and initialized.

Items with no storage class attributes default to STATIC NONLOCAL if they are EXTERNAL and not initialized, AUTOMATIC if they are INTERNAL and not initialized in a reentrant environment, and STATIC LOCAL if they are initialized or are INTERNAL in a nonreentrant environment.

The components of a structure are in the same storage class as the structure, but can not have any storage class attributes declared.

Default Boundary

Structures (including structures within structures) with no data type explicitly declared have a default boundary of WORD; where data type is specified, the default for that type is applied. Default boundaries otherwise are as follows:

Data Type	Default Boundary
FIXED(15)	HWORD
FIXED(31)	WORD
BIT	BYTE (not in a structure) bit (within a structure)
CHARACTER	BYTE
POINTER(8)	BYTE
POINTER(15)	HWORD
POINTER(16)	HWORD
POINTER(24)	WORD,2
POINTER(31)	WORD
POINTER(32)	WORD
LABEL	HWORD (may not be specified)
ENTRY	HWORD (may not be specified)

Default byte position where none is specified is 1.

Default Initialization

No initialization takes place by default. The starting values of uninitialized items are unpredictable.

Default Normality

Items are NORMAL by default.

DATA MANIPULATION

The manipulation of data may be divided into three general categories: arithmetic operations, string operations, and control operations. Arithmetic operations apply to numeric data. String operations manipulate sequences of bits or characters. Control operations determine the order in which other operations are executed.

Manipulation of arithmetic and string items is indicated in BSL by means of expressions, which employ data names, constants, and operators. An expression value may be assigned to a data item by means of the assignment statement.

Labels and entries are primarily for control operations. Several statements are defined which 'manipulate' the order of statement execution.

VALUE ASSIGNMENT

The INITIAL attribute has already been mentioned as a means of statically assigning values for data. A means of dynamically assigning values is provided in the assignment statement.

```
General form:
     receiver  =  source-expression  ;
```

The source expression is evaluated, and its value is assigned to the receiving data variable. The form of the source expression is discussed below, under Expressions. The receiver is a user's data name, subscripted if it were defined as an array, and optionally substringed (see Substring Notation) if it were defined as a string.

EXPRESSIONS

An expression may be a single data item, or it may be a combination of operators and associated data operands.

Operators

The operators available in BSL are as follows:

Operator	Priority	Normal Type	Description
+,-	1	arithmetic	prefix plus, minus
*,/,//	2	arithmetic	infix multiplication, division, remainder
+,-	3	arithmetic	infix addition, subtraction
>,<, ¬>,¬<	4	varied	infix comparisons
=,¬=,>=,<=	4	varied	infix comparisons
&	5	bit	infix and
|	6	bit	infix inclusive or
&&	7	bit	infix exclusive or

A prefix operator has a single operand, which follows the operator. An infix operand has one operand preceding it and one operand following it.

Operators are associated with operands according to priority. A low number indicates that an operator will have its operands determined before operators with higher numbers. For example, in the expression A + B * C, B and C are associated with the multiplication (since its priority number is lower than addition), and this result and A are associated with the addition. In expressions involving operators of the same priority, the operators of priority 1 are associated in order of appearance from right to left in the expression, and all others are associated within their level in order of appearance from left to right. Priority can be overridden by using parentheses to enclose operands.

Examples:

```
A+B*C*D  is equivalent to (A+((B*C)*D))
A+B-C/D  is equivalent to ((A+B)-(C/D))
(A+B)*-C is equivalent to ((A+B)*(-(C)))
```

Comparison Operators

Comparison operators may only be used in the relational expression of the IF statement. The general form of a relational expression is given with the explanation of that statement.

ARITHMETIC OPERATIONS

The arithmetic operators are the unary plus and minus, and addition, subtraction, multiplication, division, and remainder. The builtin functions ABS and ADDR (discussed under Builtin Functions) are also considered as arithmetic operators of priority level 1. In addition, the operators and, inclusive or, and exclusive or may be used in an arithmetic environment (defined further on). The result after applying any of these operators is a fullword arithmetic value.

Mixed Precisions

Use of arithmetic operands with different precisions will result in the shorter being conceptually extended on the left with the sign if it is signed, or zero if it is not signed. (FIXED variables are examples of signed values; FIXED constants and POINTER variables are examples of unsigned values.) This preserves the arithmetic value of the item, since two's complement notation is used. The actual extension may not occur, if halfword or other special length instructions can be used to achieve the equivalent effect.

Assignment Involving Mixed Precisions

If an assignment receiver's precision is greater than that of the source value, the source value is conceptually extended as above. If the receiver's precision is less, the source value is truncated on the left; this left part should just be a repetition of the sign (0 for an unsigned value).

A value assigned to a variable should be within the range of values allowed for that type of variable. Assigning a value outside the range can have unpredictable results.

Arithmetic Operations with String Items

Strings which are one, two, three, or four bytes long may be used in an arithmetic environment. They are considered as unsigned values, with all bits being significant.

STRING OPERATIONS

The operators primarily meant for strings are *and*, *inclusive or*, and *exclusive or*. They are applied to the bits making up the operands. The result of a string operation is a bit string whose length is determined from the bit length of the operands.

Operations with Unequal Lengths

For unequal lengths, the shorter string is conceptually extended on the right with zero bits. For the *and* operation, this has the effect of extending the short result with zeros to the length of the longer operand. For *inclusive or* as well as *exclusive or*, the effect is to append the difference between the longer and shorter operands to the short result.

Assignment Involving Unequal Lengths

Assignment of a string value to a shorter string receiver will truncate the source value on the right, taking the leftmost portion for the length of the receiver. Assignment of a string source value to a longer bit receiver will extend the source value on the right with zero bits to the length of the receiver. Assignment of a string value to a longer character receiver will extend the source value on the right with blanks if no operators are used, otherwise with zeros.

String Comparisons

Comparison of strings is valid only if both operands are of the same length. Character strings may be compared to bit strings, in which case the bit length of the character string is eight times its character length.

String Operations with Arithmetic Items

Arithmetic items may be manipulated as strings, as discussed under *Mixed Types*. The internal form of the item is simply treated as a string, with extension or truncation on the right as for a normal string.

MIXED TYPES

The user may mix arithmetic and string operators and operands. The type of all operations performed will be either string or arithmetic, depending on the appropriate environment. Each environment will be discussed separately below.

Subscripts and Substrings

Subscript and substring specifications are necessarily arithmetic. This is because they represent numerical information, being the array or string element position. String data in subscripts or substrings thus would always be treated arithmetically.

Assignments

If arithmetic operators appear (other than in subscript expressions) or if the receiver is FIXED or POINTER, then the evaluation of the source expression and the assignment to the receiver will be treated arithmetically. If neither condition holds, then the expression evaluation and assignment will be according to the rules for strings.

Comparisons

If arithmetic operators appear (other than in subscript expressions) or if the left operand is of arithmetic type, then the comparison will be arithmetic. If neither condition holds, then the comparison will be a string comparison.

Each comparison in a set of comparisons joined by *ands* or *ors* has this rule applied separately.

DO Terms

The control variable, and the start, increment, and terminate values of the DO statement are all considered arithmetically. The operations of setting, incrementing, and testing the control variable are all of an arithmetic nature.

Argument Expressions

String operators are not allowed in argument expressions. Arguments involving arithmetic operators are evaluated arithmetically, resulting in a fullword arithmetic value.

STATEMENTS

The statements available in BSL are described individually in this section.

The table below indicates the statement types, along with their primary function.

Statement	Function
Assignment	assigns values to a data item
CALL	provides linkage to a specified entry point
DECLARE	conveys data information to the compiler
DO	groups statements, with optional iteration
END	closes a group or procedure
ENTRY	specifies a secondary procedure entry point
GENERATE	allows insertion of assembly language text
GOTO	transfers control within a procedure
IF	provides conditional statement execution
Null	causes no action (used with ELSE clause)
PROCEDURE	specifies the primary procedure entry point
RELEASE	makes registers available to the compiler
RESTRICT	prohibits compiler use of certain registers
RETURN	returns control to the invoking procedure

<u>The Assignment Statement</u>

General form:
```
variable = expression  ;
```

The assignment statement is used to evaluate an expression and assign the resultant value to a receiving variable. The type of operations performed depends on the type of data items and operators involved, as mentioned under <u>Mixed Types</u>.

Examples:

```
A=B+C*D;
TABLE(I)=TABLE(I)+I;
```

The CALL Statement

General form:

```
CALL entry [(argument [,argument]... )] ;
```

The CALL statement provides a linkage to a specified entry point. The specified entry will be executed, with control normally returning immediately following the CALL statement. (Return of control is discussed with the RETURN statement.)

Arguments can be included as part of the linkage. An argument may be a single data item, or it may be an expression. There are some implementation restrictions on arguments; these restrictions are given in the BSL User's Guide.

Arguments are made known to the invoked entry by passing addresses in a parameter list. Arguments which are of a form suitable for being an assignment statement receiver, or which are arrays, labels or entries, have their address inserted in the list; subscripts and pointer qualifiers are evaluated as part of the address. Arguments which are constants have the address of a generated copy inserted. Arguments other than the above (i.e., expressions involving operators and/or parentheses) are evaluated, and their value is assigned to a generated temporary variable; the address of this temporary variable is inserted in the list.

A correspondence exists between the arguments of the CALL and the parameters of the entry point (indicated in the PROCEDURE or ENTRY statement). The parameter names represent the arguments in the called entry. Assigning a value to a parameter results in modifying the corresponding argument; parameters corresponding to constant arguments should not be modified.

Examples:

```
CALL NOPARAMP;
CALL ROUTINEA (ARG1,ARG2,ARG3);

CALL PROCP (ARRAY(J),I+4,32);
/*In the last example, the expression I+4 would be
evaluated and assigned to a temporary variable.
Suppose that PROCP were defined as:
     PROCP: PROCEDURE(P1,P2,P3);
Assigning to P1 would modify the corresponding element
of ARRAY; note that the subscript was evaluated at the
call, so that changing J does not change the element
of ARRAY to which P1 corresponds.  Assigning to P2 would
modify the temporary variable, but would not modify I.
Assigning to P3 would have unpredictable results, since
the corresponding argument is a constant*/
```

The DECLARE Statement

General form:

```
{DCL    }
{DECLARE} name [attribute]... [,name [attribute]...] ;
```

The DECLARE statement has already been discussed, and is the primary means of conveying information about data items to the compiler.

The DO Statement

General form:

```
   [               [BY increment  [TO terminate]]]
DO [ control=start [TO terminate  [BY increment]]];
```

The DO statement provides a statement grouping (in association with the END statement), and may specify iterative execution of the statements in the group.

In the iterative form, the control variable is set to the start value. The control is compared against the terminate value; if the BY keyword is followed by a minus (-) the control is checked for being less than the terminate value; otherwise it is checked for exceeding the terminate value. If the comparison yields true, then the iterations are terminated. If it yields false, then the statements in the group are executed. The control variable is then advanced by the increment value, and the test is made as before.

If the increment (BY) value is not specified, then it is assumed to be 1. If the terminate (TO) value is not specified, then the loop is potentially infinite, requiring some other exit mechanism. If both the increment and terminate values are absent, then the group is executed once, setting the control to the start value before executing.

The start, increment, and terminate values may all be expressions.

Modification within the loop of the control variable will be reflected in subsequent iterations. Variables involved in the increment and terminate expressions must not be changed in the loop; such changes have unpredictable effects on incrementation and testing.

Examples:

```
DO;                   /*no iteration specified*/
DO I=1;               /*no iteration specified*/
DO V=START TO FINI;   /*iteration with increment 1*/
DO SV=0 BY 2 TO 10;   /*iteration specified*/
DO REPT=1 BY 1;       /*requires exit mechanism*/
DO I=10 BY -1 TO 1;   /*test is for I less than 1*/
```

The END Statement

General form:

```
        ┌ label       ┐
    END └ entry-name  ┘ ;
```

The END statement indicates the end of the statements in a group or procedure. With no label or entry name following, it closes out the nearest preceding unclosed group or procedure. With a following label, which must be from a preceding unclosed DO statement, it serves as an end for all unclosed groups up to and including the one started by that DO statement. (This effect is called multiple closure.) An entry name following the END keyword must be the name of the nearest unclosed procedure, thus serving as a check on matching PROCEDURE and END statements.

An END statement which ends a procedure, and which is encountered in the execution path of that procedure, will act as a RETURN for that procedure.

Examples:

```
END;        /*closes nearest group or procedure*/
END DOSET;  /*closes all up to DOSET*/
```

The ENTRY Statement

General form:

```
    ENTRY [ (parameter [,parameter]... )] ;
```

The ENTRY statement specifies a secondary entry point for a procedure. It is preceded by an entry name by which this entry point is known. A correspondence exists between the arguments of the invocation and the parameters of the entry point, as discussed with the CALL statement. Parameters common to several entry points must have the same position in the parameter list.

Examples:

```
EP:         ENTRY(PAR1,PAR2);
EPNOPAR:    ENTRY;
```

The GENERATE Statement

General form:

```
{ GEN      }
{ GENERATE } [ (assembler-text) ]  ;
```

The GENERATE statement provides a means of inserting assembly language text into BSL generated code. Use of this facility requires knowledge of compiler code and data generation characteristics. In the form with text in parentheses, the text is mapped starting at column 10 of an output image, and must be on a single line. The end of text is indicated by the sequence: right parenthesis, optional blanks, and semicolon. A label on the statement would be placed in the name field of the output card.

In the form with parenthesized text absent, any labels on the GENERATE statement are first put out into the assembler text with an 'EQU *'. The rest of the input card is ignored. Columns 1 through 72 of subsequent input cards are put out into the assembler text (with sequencing in columns 73 through 80), until a delimiting control card is encountered. This control, $ENDGEN, is discussed in the BSL User's Guide. This type of GENERATE statement is called a block GENERATE, since more than one assembly statement can be inserted.

Names defined in the assembler text included by a GENERATE are not known by the compiler, since it does not analyze these statements. To make such names known so that conflicting definitions are not produced, the user would declare them with the attribute GENERATED in addition to their data attributes. Such items are assumed by the compiler to be addressable, and it is the user's responsibility to insure this.

Examples:

```
     GENERATE (COPY  SECTION   );
L:   GEN (LPSW MYOWNPSW );
     GENERATE ;
     SIO  R7
     BC   1,LA
     BC   2,LB
     BC   4,LC
$ENDGEN
     DCL DCBISAM GENERATED CHAR(200);
     GENERATE;   the rest of this card is ignored
DCBISAM DCB ...  this card is generated
            ...  this card is generated
$ENDGEN          the rest of this card is ignored
```

The GOTO Statement

General form:

```
{ GOTO  }
{ GO TO } label ;
```

The GOTO statement provides a transfer of control to the statement indicated by the specified label. Transfers across procedure limits may cause incorrect results since no procedure initiation or completion is implied in the GOTO statement. Transfers into iterative DO groups may also cause incorrect results, since no adjustment to the loop control is implied.

Examples:

```
GOTO L;
GO TO CONTIN;
GOTO INDIRECT; /*INDIRECT might be a BASED LABEL,
                 thus providing indirect transfers*/
```

The IF Statement

General form:

```
IF relational-expression THEN unit-1 [ELSE unit-2]
```

The IF statement provides conditional statement execution, dependent on the validity of a relational expression. The general form of a relational expression is given below:

```
operand comparison-op operand [ { & } operand comparison-op operand ] ...
                                { | }
```

The comparison operands may be expressions; if they involve _and_, _inclusive or_, or _exclusive or_ then they must be enclosed in parentheses. _And_ and _or_ can be used in a relational expression to connect a set of comparisons; _and_ indicates that all of the connected comparisons must be true, while _or_ indicates that any of the connected comparisons must be true. Comparison operations cannot be enclosed in parentheses.

If the relational expression is true, then unit 1 is executed; unit 2, if present, will be skipped. If the relation is false, then unit 1 is skipped; execution flow continues with unit 2 if present, or otherwise with the next statement.

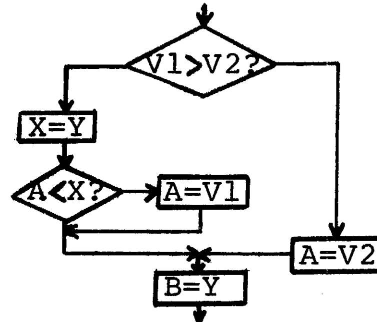
Each unit is either a group of statements (defined by the DO - END bracketing) or a single statement other than DECLARE, DO, END, ENTRY, PROCEDURE, RESTRICT, or RELEASE. Where the unit is another IF statement, any ELSE clauses will be applied starting at the innermost IF. An ELSE clause, if there is one, must immediately follow unit 1.

Examples:

```
IF A=B THEN GOTO SKIPIT;
IF FLAG='0'B THEN R=3; ELSE R=-3;

IF V1>V2 THEN
    DO;
    X=Y;

    IF A<X THEN A=V1;
    END; /*no ELSE for IF above*/
ELSE A=V2;
B=Y;
```



The Null Statement

General form:

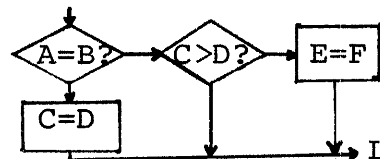
```
;
```

The null statement causes no action. One use of a null statement is as an ELSE unit to achieve proper association of ELSE clauses and IF conditions.

Examples:

```
     ;
CONTINUE:   ;

     IF A=B THEN IF C>D THEN E=F;
                        ELSE ;
             ELSE C=D;
     GOTO L;
```



The PROCEDURE Statement

General form:

```
    { PROC      }
    { PROCEDURE } [ (parameter [,parameter]... ) ] [OPTIONS
                  (option[,option]...) ];
```

The PROCEDURE statement specifies the primary entry point for a program section. It is preceded by an entry name by which this entry point (and procedure) is known. A correspondence exists between the arguments of the invocation and the parameters of the

entry point, as discussed with the CALL statement. Procedures can be nested, in which case contained procedures can be invoked only from the immediately containing procedure or from other procedures within the containing procedure. A procedure is syntactically completed by an END statement.

Any internal procedures must be defined immediately before the END statement for the containing procedure. These procedures may themselves have internal procedures, in the same format.

The user may control the entry and exit code produced by the compiler, using the OPTIONS field of the PROCEDURE statement. These options are discussed in a separate section on _Procedure Options_.

Examples:

```
MAINP:  PROCEDURE(PARAM);
        DCL INTP INTERNAL ENTRY;
        CALL INTP;
        RETURN;
INTP:   PROCEDURE;
        RETURN;
        END INTP;
        END MAINP;
```

The RELEASE Statement

General form:

```
RELEASE (register [,register]... ) ;
```

The RELEASE statement makes the indicated registers available to the compiler in generating code. The user may still explicitly reference the registers, but he should be aware of possible effects both on and by the compiled code.

The registers which may be released are discussed under the topic of _Register Usage_. Certain registers are preassigned and cannot presently be released or restricted.

At the start of each procedure, including each internal procedure, registers are assumed available for use by the compiler. Declaration of an item as register does not restrict that register.

Example:

```
RELEASE (4,7,9);
```

The RESTRICT Statement

General form:

```
RESTRICT (register [,register]... ) ;
```

The RESTRICT statement prohibits the compiler from using the indicated registers. The registers will be used only if explicitly referenced by the programmer.

The registers which may be restricted are discussed under the topic of Register Usage. Certain registers are preassigned and cannot be restricted or released.

At the start of each procedure, including each internal procedure, registers are assumed available for use by the compiler; this includes registers which were restricted in the outer procedure. Declaration of an item as register does not restrict that register.

Example:

```
RESTRICT (2,3);
```

The RETURN Statement

General form:

```
RETURN  [TO label] ;
```

The RETURN statement terminates execution of the procedure in which it is contained, returning control to the invoking procedure. In the form with no return label, it returns control immediately past the point of invocation. An END statement terminating a procedure will also serve as a return if control ever reaches the statement.

In the form with a return label, the normal return activity is performed, except that control is returned to the point in the calling procedure indicated by the label. The label should be in the invoking procedure, and known in the returning procedure.

Examples:

```
P:   PROCEDURE(L,M);
     DCL L LABEL, XL LABEL EXTERNAL,
          Q POINTER EXT, DL LABEL BASED(Q);
     RETURN;    /*returns to caller past point of call*/
     RETURN TO L; /*returns to label indicated by first
                    parameter*/
     RETURN TO XL; /*returns to label XL, which must be
                     defined in caller as LABEL LOCAL
                     EXTERNAL*/
     RETURN TO DL; /*returns through pointer Q, which
                     must have been set in caller*/
     END P;     /*this would act as a simple return if
                  control ever reached this point*/
```

COMPILE TIME FACILITIES

The BSL user can modify his program text at the time it is being compiled. This function is achieved by considering compile time as a two stage process. The first stage, or macro stage, scans the user's source text and forms a modified source text. A set of macro statements tells the compiler how to modify the source text, but these macro statements will not themselves be included in the modified text. The modified text serves as input to the second stage, which is the normal compilation stage.

Some uses of the compile time, or macro, facilities are indicated below:

- source program parameterization.
- conditional compilation of source text segments.
- inclusion of text strings residing in a user library.

BASIC STRUCTURE

Macro Statements

Macro statements are recognized by a leading percent sign (%). A macro statement is executed when it is encountered in the scan of source text. This execution may affect the point at which scanning continues, and may also affect the subsequent modification of source text. The macro statements available in BSL are indicated below, along with their primary function.

Statement	Function
ACTIVATE	marks a macro variable as replaceable in source text.
Assignment	assigns a value to a macro variable.
DEACTIVATE	marks a macro variable as not replaceable in source text.
DECLARE	conveys attribute information for macro variables.
GOTO	changes the point at which scanning continues.
IF	provides conditional macro statement execution.
INCLUDE	includes text strings from a user library.
Null	causes no action. It can specify a GOTO target.

Macro statements may not appear on the same input line with non-macro source text. A macro statement may be spread across a number of input lines, and a number of macro statements may appear on the same line.

Examples of macro statements:

```
%DECLARE (A,B,C) FIXED;        /*Macro DECLARE*/
%A=4; %B=2*A                   /*Macro assignments*/
%GOTO L;                       /*Macro GOTO*/
%L:  ;                         /*Macro null, used as a GOTO target*/
%C=3; Z=8;                     /*Illegal appearance of macro and
                                 non-macro text on the same line.
                                 The assignment Z=8 is not a macro
                                 statement since it does not have
                                 a leading percent sign*/
```

Macro Variables

A macro variable is simply a variable which exists during the macro stage of compilation. It is defined by its appearance in a macro DECLARE statement.

Macro variables may be of type FIXED or of type CHARACTER. These types are somewhat different from the non-macro FIXED and CHARACTER types. No size may be specified for FIXED; a standard size is used. No length may be specified for CHARACTER; a macro string's length is variable, being equal to the length of the most recent value assigned to the string.

Examples:

```
%DECLARE CMAC CHARACTER, FMAC FIXED;
%DECLARE F FIXED(31);     /*Illegal; a size is not allowed*/
%DECLARE C CHARACTER(2);  /*Illegal; a length is not allowed*/
%FMAC=3;
%UNKNOWN=3;               /*Illegal unless UNKNOWN appeared
                            in a macro DECLARE statement*/
%CMAC='ABCD';             /*CMAC receives a length of four*/
```

Source Text Replacement

Modification of source text takes place by replacing the appearance of a macro variable's name by its value. For a FIXED item, this replacement value is a character string representing the value of the item as a decimal number, preceded by a minus sign if it is negative. A replacement value is not enclosed by quote marks when substituted into the source text for the macro name.

A blank is appended to each end of the replacement value when it is substituted into the source text. This means that identifiers and composite delimiters, which cannot have embedded blanks, cannot be formed from two parts, one part in the unsubstituted source text and the other part in the replacement value. The replacement value cannot contain macro statements, nor should it contain unmatched comment or string delimiters.

A macro name appearing within a comment or a string constant will not be replaced. A name will not be replaced if it is preceded or followed in text by a letter, digit, or quote mark; for example, the macro name B will not be replaced in the strings AB, 101B, or '1'B. A name must be marked as available for replacement; this will be the case unless the user indicates otherwise, as discussed in the section on <u>Macro Activation</u>.

Macro statements will not appear in the modified source text. A macro statement is replaced by blanks. Comments associated with a macro statement are also replaced by blanks.

Examples:

```
%DECLARE(A,B)FIXED,
  (COND, IS, PLUS) CHARACTER;
%A=7; %B=-42;
%COND='='; %IS='='; %PLUS='+';
V=A+B;               /* is modified to V= 7 + -42 ; */
T IS V PLUS B;       /* is modified to T  =  V  +  -42 ; */
Z = '1'B;            /* is unmodified, since B is preceded
                        by a quote */
IF W ¬COND A ...     /* is modified to IF W ¬ = 7 ... which
                        is incorrect because of the blank
                        between ¬ and = */
```

<u>Rescanning</u>

Replacement of a macro name results in a new segment of source text. This new segment of text is scanned to see if it contains further references to macro variables. Such references will have replacements and rescans performed, until no further replacements can be made. For FIXED macro variables, the replacement value will not include any names, and therefore is not rescanned.

The value of a macro variable should not contain a reference to the name of the macro variable. Rescanning such a value would again require substituting the value, resulting in a replacement loop. This situation can also arise indirectly if some other replacement in the rescan references the macro name.

Examples:

```
%DECLARE (C,D, RESCAN, LOOP, INDIRECT, LOOPING) CHARACTER;
%C='CVALUE'; %D='DVALUE';
%RESCAN='C+D';              /* The appearance of RESCAN in source
                               text will result in a substitution
                               value of CVALUE + DVALUE*/
%LOOP='C+LOOP';             /* The appearance of LOOP is illegal,
                               since the rescan would reference
                               LOOP*/
%INDIRECT='D+LOOPING';
%LOOPING='C+INDIRECT';      /* The appearance of INDIRECT or
                               LOOPING would be illegal, since
                               a rescan would eventually result
                               in a replacement loop*/
```

MACRO STATEMENTS

Macro statements are discussed in this section. The subsections, and the statements discussed in each, are indicated below.

Subsection	Statements
Macro Declaration	DECLARE
Value Assignment	Assignment
Scan Control	GOTO, Null
Text Inclusion	INCLUDE
Conditional Execution	IF
Macro Activation	ACTIVATE,DEACTIVATE

Macro Declaration

General form of a macro DECLARE statement:

```
 { DCL     }  { (name   [, name] ...) }   { FIXED          }
%{ DECLARE }  {  name                 }   { { CHAR      }  }
                                          { { CHARACTER }  }

        [   { (name   [, name] ...) }   { FIXED          } ]
        [ , {  name                 }   { { CHAR      }  } ]       ;
        [                               { { CHARACTER }  } ]...
```

All macro names must be declared before any reference to them. There is no default type for macro variables. No initial value (or length) can be assumed for a macro variable; it will have a value (and length) only after its appearance in a macro assignment statement.

A macro DECLARE statement must be encountered in the source text scan in order to be recognized. A macro DECLARE in text which is not scanned (because of a macro GOTO past it, for example) is ignored.

Examples:

```
%DECLARE C CHARACTER, F FIXED;
%DECLARE (C1,C2)CHAR,(V1,V2,V3) FIXED;
%DCL FMAC FIXED;

%GOTO ML;
%DECLARE UNKNOWN CHAR;          /*Scanning bypasses this macro
                                  DECLARE.  It is ignored*/
%ML:  ;
```

Value Assignment

General form of a macro assignment statement for CHARACTER macro variables:

```
                             {constant} [  {constant} ]     ;
%[label:]...variable =       {variable} [||{variable} ]...
```

General form of a macro assignment statement for FIXED macro variables:

```
                          [+] {constant} [    [+] {constant} ]
%[label:]...variable =    [-] {variable} [op  [-] {variable} ];
```

The macro assignment statement assigns a value to a macro variable. It only allows assigning a source value of the same type as the target variable.

For CHARACTER assignments, the operands in the source expression may be either character constants or macro variables of type CHARACTER. The target variable receives both the value and the length associated with the source expression. The concatenation operation (||) may be used in the source expression. This operator simply connects its operands together, forming a string with a length equal to the sum of the lengths of the operands.

For FIXED assignments, the operands in the source expression may be either decimal constants or macro variables of type FIXED. A maximum of two terms is allowed in the source expression. The operators allowed are +,-,*,/, and //.

Examples:

```
%DECLARE (F1,F2,F3,F4) FIXED,
         (C1,C2,C3,C4) CHARACTER;
%F3 = 13;%F2 = 7;
%F1 = F2 + F3;
%F4 = -F1 * F3;
%F1 = (F1 + F3);    /*Illegal.  Parentheses are not allowed*/
%F1 = F1 + F2 + F3; /*Illegal.  More than two terms in the
                      source expression*/
%C4 = 'FBCD';       /*C4 receives a length of four*/
%C3 = '-39';        /*C3 receives a length of three*/
%C1 = C3;           /*C1 receives the value and length of C3*/
%C2 = C4||C1;       /*The values of C4 and C1 are concatenated,
                      resulting in the string 'FBCD-39'.  This
                      value is assigned to C2, which receives
                      a length of seven*/
%C2 = C1 & C3;      /*Illegal.  The 'and' operator is not
                      allowed*/
%C2 = C1 + C3;      /*Illegal.  Strings may not be used
                      arithmetically*/
```

Scan Control

General form of a macro GOTO statement:

```
%[label:]... GOTO label ;
```

General form of a macro null statement:

```
%[label:]... ;
```

The macro GOTO statement alters the sequence of scanning source text, causing continuation of macro activity at the macro statement having the specified label. The label must be the label of a macro statement further on in the text. Scanning will resume after the transferred-to statement, unless that statement also modifies the scanning sequence.

Examples:

```
%GOTO L1;
/*this text is skipped*/
%L1:  ;
/*this text is scanned*/
%GOTO L1;                /*Illegal.  L1 has already appeared,
                           and cannot be transferred to*/
```

Text Inclusion

General form of a macro INCLUDE statement:

```
%[label:]... INCLUDE ddname (membername) ;
```

The macro INCLUDE statement is used to incorporate text records on a user data set into the source text. This incorporated text is scanned in the same manner as the source text, replacements being made and macro statements being executed. When scanning comes to the end of the incorporated text, it continues following the INCLUDE statement.

The incorporated text may include macro statements as well as program text. The only macro statement which cannot appear is another INCLUDE statement. The incorporated text cannot have unmatched comment or string delimiters.

The meaning of 'ddname' and 'membername' is as for the $INCLUDE compiler control option, which is discussed in the BSL User's Guide. The $INCLUDE may still be used. The macro INCLUDE incorporates text during macro activity, while $INCLUDE incorporates text after macro activity and does not submit the text to scanning for macro substitution.

Examples:

```
%INCLUDE COMMON (DECLARES);
%INCLUDE COMMON (MACROS);
```

Conditional Execution

General form of a macro IF statement:

```
%[label:]... IF relational-expression %THEN unit-1
             [%ELSE unit-2]
```

The macro IF statement conditionally executes other macro statements. Flow through the macro IF statement is as for the non-macro IF statement. The units for %THEN and %ELSE may be any macro statement except DECLARE or IF.

General form of a CHARACTER relational expression:

```
                         {constant}
variable relational-op   {variable}
```

General form of a FIXED relational expression:

```
                                  [+] {constant} [   [+] {constant} ]
     variable relational-op       [-] {variable} [ op[-] {variable} ]
```

Only macro values of the same type may be compared. For CHARACTER comparisons, both operands must be of the same length. The relational operators are =,¬=,>=,<=,¬>, and ¬<.

Examples:

```
%DECLARE(A,B) FIXED;
%A=7; % B=13;
%IF A=8% THEN %B=B/2;
%IF B>A*2% THEN %GOTO L1;
/*This text will be skipped if the stated relationship is true*/
%L1:  IF A>B %THEN %INCLUDE LIB(VERSIONA);
      %ELSE %INCLUDE LIB(VERSIONB);
/*Scan will continue here after processing the appropriate
  INCLUDE unless the INCLUDE contains a transfer to some
  other scan point*/
%IF A=7 %THEN %DECLARE C CHAR;      /*Illegal.  An IF unit cannot
                                      be a DECLARE statement*/
%IF A=7 %THEN %IF B=6 %THEN %B=12; /*Illegal.  An IF unit cannot
                                      be another IF statement*/
```

Macro Activation

General form of the macro ACTIVATE and DEACTIVATE statements:

```
                   { {ACT       } }
                   { {ACTIVATE  } }
%[label:]...       { {DEACT     } }    name   [,name]...   ;
                   { {DEACTIVATE} }
```

The ACTIVATE and DEACTIVATE statements are used to control whether or not a text reference to a macro variable name is replaced by the value of the macro. This function is useful, for example, where an included macro name conflicts with a name used in the program. The user can deactivate a macro variable so that the name is not replaced when it appears in text. At some later point in the text a macro variable could be reactivated, so that a reference to its name would again be replaced by its value.

A macro variable which has been deactivated is still available for use within macro statements. Its value is not altered by the fact that it has been deactivated.

The declaration of a macro variable serves as an activation of that name. A name needs to be activated only if it has been deactivated and the user again wishes to use it for replacement.

Examples:

Original Text	Modified Text
%DECLARE A FIXED;	
%A=4;	
V1 = A;	V1 = 4 ;
%DEACTIVATE A;	
V2 = A;	V2 = A;
%A = 5;	
%ACTIVATE A;	
V3 = A;	V3 = 5 ;

ADDITIONAL TOPICS

SUBSTRING NOTATION

It is often desirable to reference only a portion of a string variable. The notation for such a substring reference is similar to the array subscript notation. For the simple case in which one bit or character is desired, the position of that element is specified, by a data constant or variable, in parentheses following the name of the original string item. Where a string of elements is desired, values indicating the first and last element positions are placed in the parentheses, separated by a colon. If the string is dimensioned, the subscript appears first in the parentheses, separated from the substring indicator by a comma.

Examples:

```
DECLARE STRING(100) CHAR(256), FLAGS BIT(8);

   FLAGS             refers to the entire 8-bit string
   FLAGS(3)          refers to the third bit in FLAGS
   FLAGS(3:6)        refers to the third through sixth
                       bits of FLAGS
   STRING(4)         refers to the fourth 256-character
                       string in the array STRING
   STRING(7,9)       refers to the ninth character in
                       the seventh 256-character string
   STRING(7,2:5)     refers to the second through fifth
                       characters in the seventh
                       256-character string
   STRING(I,J)       refers to the Jth character in the
                       Ith 256-character string
```

Variable Length Substrings

Where both element positions are specified and at least one is a variable, the substring has a variable length. Variable length substrings may not be used in an arithmetic environment. A variable length substring as an assignment receiver must not be longer than the source; in an assignment source expression, it must not be longer than the receiver (unless the receiver is also of variable length). As an operand of a comparison, it should not be used unless the other operand is also a variable length substring of the same length.

Examples:

```
STRING(1:I)='ABC';            /*I must be 1, 2, or 3*/
STRING(3:33)=STRING(103:K); /*K must be no less than 103,
                              but no greater than 133*/
STRING(I:J)=STRING(2:8);    /*J-I+1 must be no less than 1,
                              but no greater than 7*/
```

Bit Substrings

A number of restrictions exist on substringing bit items. A single position substring may not be specified by a data variable. Where both element positions are given and at least one is a variable, the first must be that of the leftmost bit in some byte and the second must be that of the rightmost bit in some (possibly different) byte. More precise restrictions are given in the BSL User's Guide.

Examples:

```
DCL B BIT(24);
B(2:13)='100000000111'B;  /*legal*/
B(1:J)='FFFFFFFF'X;  /*J must be 8, 16, or 24*/
B=B(I:J);  /*I must be 1, 9, or 17; J must be
           8, 16, or 24, and greater than I*/
```

BUILTIN FUNCTIONS

A number of basic functions are defined in BSL. References to these functions consist of the appropriate identifier followed by an argument or arguments in parentheses. Each function has a value which depends on the arguments.

ABS Builtin Function

General form: ABS (expression)

The ABS function has one argument, which is an expression. It returns an arithmetic value which is the absolute value of the argument.

Examples:

```
V=V+ABS(A-B);
XS=ABS(ABS(A-B)-ABS(C/D));
```

ADDR Builtin Function

General form: ADDR (data-item-name)

The ADDR function has one argument, which is the name of a data item. (This includes array, structure, label and entry names, as well as array element references.) It returns a pointer value which is the initial storage location for that item. This function, when applied to based items, yields a value determined from the associated pointer. The ADDR function may not be applied to items in REGISTER class.

Examples:

```
PTOVAR=ADDR(VAR);
PELEM3=ADDR(ELEM(3));
DCL P PTR INIT(ADDR(V)+2);
```

POINTER ASSOCIATION

A BASED item may have a location associated at a reference to the item by preceding the reference with a pointer followed by the combination ->. This notation is called pointer qualification. The pointer will be used to locate the item, overriding any pointer supplied in the declaration of the BASED item.

A component of a BASED structure is also BASED. A pointer preceding such an item is treated as pointing to the start of the outermost structure, and not directly at the component. The location is determined by combining the item's offset from the outermost structure with the pointer indicated; this offset does not change the value of the pointer, serving only to locate the item correctly.

Examples:

```
DECLARE  1  FIELD   BASED(P),
            2  ELEM  FIXED,
            2  FLAG  BIT(8),
         (P,Q) POINTER;
ELEM=Q->ELEM;   /*The receiver refers to ELEM as
                    located by P pointing to FIELD.
                  The righthand refers to ELEM as
                    located by Q pointing to FIELD*/
Q->FLAG=FLAG;   /*The receiver refers to FLAG as
                    located by Q pointing to FIELD.
                  The righthand refers to FLAG as
                    located by P pointing to FIELD*/
```

Multiple levels of pointer qualification may also be used. The intermediate level qualifiers must be both POINTER and BASED.

Examples:

```
DECLARE (P,Q) POINTER,
        R POINTER BASED(Q),
        V FIXED BASED(P);
V=R->V;      /*The receiver refers to V as
                 located by P.
               The righthand refers to V as
                 located by R, which itself
                 is located by Q*/
P->R->V=0;   /*The receiver refers to V as
                 located by R, which itself
                 is located by P*/
```

REGISTER USAGE

A user wishing to explicitly reference registers should be aware of the register conventions employed in the System/360 code produced by the compiler, which are explained in the following paragraphs. All registers may be referenced and used in BSL, although some have special preassigned functions.

Each procedure, including each internal procedure, employs a standard save mechanism, with register 13 as the area pointer and register 14 as the return location pointer. Register 15 is used in linking to external procedures, and is assumed available for use in the entry code. Register 11 is used for code addressing, unless it is overridden by a CODEREG option (see Procedure Options). Register 12 is used for data addressing in a REENTRANT environment, unless it is overridden by a DATAREG option (see Procedure Options). Register 1 points to the parameter list if the procedure has parameters or if arguments are passed to a called entry.

Registers 14 and 15 are used by the compiler for most arithmetic calculations, with register 0 involved in certain cases. Register 10 is also occasionally required for complicated data moves. The other registers are used as available for indexing, external item addressing, and pointer manipulation.

The registers which cannot be restricted or released by the user are 13, 14, 15, 0, 10, any code registers, and any data registers. Register 1 also cannot be restricted or released if the procedure has parameters passed to it. Registers 11 and 12 may be restricted and released if they are not used as code or data registers. Any of the other registers, if they have no assigned function, may be restricted and released. Registers which are not restricted are assumed available for use by the compiler in the generated code.

Since registers are saved on entry to and restored on exit from a procedure, changes to registers normally do not propagate back to the calling procedure except possibly by modifying the save area. In the next section, procedure options are provided for controlling the saving and restoring of registers so that changes can be propagated back when desired.

PROCEDURE OPTIONS

The user is provided a means of controlling the prolog and epilog activities generated by the compiler. By means of options specified for a procedure, the user may decrease the overhead of entry and exit, and may even eliminate it.

The CODEREG Option

General form: CODEREG(register [,register]...)

The CODEREG option indicates that the specified registers are to be used for code addressability. (Register 11 will be used if no CODEREG is specified.) Each register provides 4095 bytes of addressability.

If register 0 is specified, no addressability is established, and it is assumed that the user will insure his own addressability (for example, through GENERATEs). Prolog and epilog code may depend on code addressability; the user should be aware of compiler addressability requirements.

CODEREG may not be specified for an internal procedure, since its addressability is part of that of the external procedure.

The DATAREG Option

General form: DATAREG(register [,register]...)

The DATAREG option indicates that the specified registers are used for data addressability. (Register 12 will otherwise be used in a REENTRANT environment; no data register will be used in a non-REENTRANT environment). Each register provides 4095 bytes of addressability.

If register 0 is specified, no addressability is established, and it is assumed that the user will insure data addressability or that there is no data that needs to be addressable separately from the code. Prolog and epilog code may depend on data addressability; the user should be aware of compiler addressability requirements.

DATAREG may not be specified for an internal procedure, since its data area is part of a general data area provided for all procedures by the external procedure.

The REENTRANT Option

General form: REENTRANT

The REENTRANT option indicates that the compiler should generate reentrant code, and that it should dynamically acquire data storage for data areas defined as AUTOMATIC. Save areas and compiler work areas would also be dynamically acquired. This attribute presently has meaning primarily in an OS/360 environment, in which it is implemented as a register form of GETMAIN in the external procedure's prolog with a corresponding FREEMAIN in the epilog.

The REENTRANT option will cause register 12 to be used for data addressability unless the DATAREG option is also specified. REENTRANT may not be specified for internal procedures.

The SAVE Option

General form: SAVE [(register [,register]...)]

The SAVE option indicates which registers are to be saved and restored by a procedure. If no registers are specified, then all registers are saved. (This is the standard action.) If registers are specified, then only those registers are saved and restored. Other registers will have their then current value on exit.

SAVE may be specified for internal procedures.

The DONTSAVE Option

General form: DONTSAVE [(register [,register]...)]

The DONTSAVE option indicates which registers are not to be saved and restored by a procedure. If no registers are specified, then no registers are saved. If registers are specified, then those registers will have their then current value on exit. Other registers will have their value restored to the original entry value on exit.

DONTSAVE may be specified for internal procedures. DONTSAVE may not be specified if SAVE is specified.

The NOSAVEAREA Option

General form: NOSAVEAREA

The NOSAVEAREA option indicates that no save area is to be generated for this procedure. (This is useful, for example, for procedures not invoking any other functions.) This also eliminates the save area chain updating.

NOSAVEAREA may be specified for internal procedures.

Combining Options

The combination of the options NOSAVEAREA, DONTSAVE, CODEREG(0), and DATAREG(0) results in the complete absence of any prolog code in the object program, and the absence of epilog code except for a branch on register 14.

Combining the options REENTRANT and DATAREG(0) will produce a DSECT for all AUTOMATIC data, but will not produce the GETMAIN, addressability, or FREEMAIN involving that area. Initialization of compiler work areas is suppressed, and must be provided by the user; this is discussed in the BSL User's Guide.

NAME PLACEHOLDER

A structure name is often specified simply because of the requirement for a name on the structure; the name itself is not significant (except, perhaps, as documentation) and is never referenced. Similarly, a component name may not be significant, merely serving to name a filler item which adjusts the mapping of subsequent components.

Where the name of a structure or component is not significant, it may be replaced by an asterisk (*). The asterisk serves in place of the name, not requiring the user to provide distinct names. The asterisk may not be used for items which are not structures or components of structures.

Examples:

```
DECLARE 1 *,
          2 ITEM1,
          2 ITEM2,
          2 * CHAR(3) /*FILLER*/,
          2 BYTE CHAR(1);
DECLARE * CHAR(256);          /*Illegal.  The asterisk is not
                                in a structure or component
                                name position*/
```

VARIABLE PARAMETER LISTS

At a call, a parameter list will be produced with one word for each argument given. (If no arguments are given, then no parameter list is produced.) No indication is normally given of the last word in the list.

The end of the list will be indicated if the called entry were declared with the OPTIONS (VLIST) attribute. This attribute will cause the high order bit of the last word in the list to be set to '1'B; the bit would normally be '0'B. The called entry can then include checks for this bit.

Examples:

```
DECLARE  E  ENTRY,
            EVL ENTRY OPTIONS(VLIST);
CALL E(A,B);    /*A parameter list of two words is provided.
                  The high order bit of both words will be
                  '0'B*/
CALL EVL(A,B); /*A parameter list of two words is provided.
                  The high order bit of the first word will
                  be '0'B.  The high order bit of the second
                  (and last) word will be '1'B*/
CALL EVL;       /*No parameter list is produced*/
```

The number of parameters specified at an entry point should be the number of arguments passed; high order bits of the parameter list words are assumed to be '0'B. When these conditions are not true (as would be the case where OPTIONS(VLIST) was declared on the calling side), the user should manipulate the parameter list himself using register 1 (the parameter list register); he should not specify a list of parameters on the PROCEDURE or ENTRY statement. Further details on manipulating the parameter list can be found in the BSL User's Guide.

APPENDIX I: LANGUAGE KEYWORDS

Status	Keyword	Use	References
	ABNL	data attribute	28
	ABNORMAL	data attribute	28
R	ABS	builtin function	32,56
R	ADDR	builtin function	25,32,56
	AUTO	data attribute	22,23
	AUTOMATIC	data attribute	22,23,30,60,61
	BASED	data attribute	16,17,22,24,28,29,57
	BDY	data attribute	26
	BIT	data attribute	15,29,30,55
	BOUNDARY	data attribute	26
R	BY	iteration term	38
	BYTE	boundary choice	26,30
R	CALL	statement header	36,37,39,43,63
	CHAR	data attribute	15
	CHARACTER	data attribute	15,29,30
	CODEREG	procedure option	59,61
	DATAREG	procedure option	59,60,61
R	DCL	statement header	14,38
R	DECLARE	statement header	14,36,38,42
R	DO	statement header	12,35,36,38,39,41,42
	DONTSAVE	procedure option	60,61
	DWORD	boundary choice	26
R	ELSE	false path header	36,41,42
R	END	statement header	12,13,36,38,39,43,44
R	ENTRY	data attribute	17,29,30
R	ENTRY	statement header	11,13,29,36,37,39,42,63
	EXT	data attribute	20
	EXTERNAL	data attribute	20,22,28,29,30
	FIXED	data attribute	15,29,30,33,35
R	GEN	statement header	40
R	GENERATE	statement header	36,40,59
	GENERATED	data attribute	22,23,40
R	GO TO	statement header	29,41
R	GOTO	statement header	29,36,41,45,48,49,50
	HWORD	boundary choice	26,30
R	IF	statement header	11,32,36,41,42
	INIT	data attribute	27
	INITIAL	data attribute	27,31
	INT	data attribute	20
	INTERNAL	data attribute	20,23,29,30
	LABEL	data attribute	17,29,30
	LOCAL	data attribute	22,27,30
	NONLOCAL	data attribute	22,23,30
	NORMAL	data attribute	28,30
	NOSAVEAREA	procedure option	61

R - reserved identifier

APPENDIX I: LANGUAGE KEYWORDS (continued)

Status	Keyword	Use	References
	OPTIONS	options header	42,43,59
	OPTIONS	data attribute	63
	POINTER	data attribute	16,29,30,33,35,57
R	PROC	statement header	42
R	PROCEDURE	statement header	11,13,29,36,37,39,42,43,63
	PTR	data attribute	16
	REENTRANT	procedure option	23,59,60,61
	REG	data attribute	22,24
	REGISTER	data attribute	22,24,56,58
R	RELEASE	statement header	36,42,43
R	RESTRICT	statement header	36,42,44
R	RETURN	statement header	36,37,39,44
R	RETURN TO	statement header	44
	SAVE	procedure option	60
	STATIC	data attribute	22,27,30
R	THEN	true path header	41
R	TO	iteration term	38
	VLIST	option choice	63
	WORD	boundary choice	26,30

COMPILE TIME KEYWORDS

Status	Keyword	Use	References
%R	ACT	statement header	52
%R	ACTIVATE	statement header	45,48,52
	CHAR	data attribute	48,49
	CHARACTER	data attribute	46,48,49,52
%R	DCL	statement header	48
%R	DEACT	statement header	52
%R	DEACTIVATE	statement header	45,48,52
%R	DECLARE	statement header	45,46,48,49,51
%R	ELSE	false path header	51
	FIXED	data attribute	46,47,48,49,52
%R	IF	statement header	45,51
%R	INCLUDE	statement header	45,51
%R	THEN	true path header	51

R - reserved identifier
%R - reserved identifier in compile time statements

APPENDIX II: ATTRIBUTE CONFLICTS

The matrix below indicates conflicts between attributes. An X in a position indicates that the attributes conflict. A number in a position indicates that the appropriate note below applies.

	ABNORMAL	AUTOMATIC	BASED	BIT	BOUNDARY	CHARACTER	Component	Dimension	ENTRY	EXTERNAL	FIXED	GENERATED	INITIAL	INTERNAL	LABEL	LOCAL	NONLOCAL	NORMAL	OPTIONS	Parameter	POINTER	REGISTER	STATIC	Structure
ABNORMAL	X						X		X						X			X	X					
AUTOMATIC		X	X				X		X	X		X	X		X	X	X		X	X		X	X	
BASED		X	X				X			X		X	X			X	X			X		X	X	
BIT				X		X			X		X				X				X			X		
BOUNDARY					X				X						X				X			X		
CHARACTER				X		X			X		X				X				X		X	X		
Component	X	X	X				X	1	X	X		X		X	X	X	X	X	X	X		X	X	
Dimension							1	X	X						X				X			X		
ENTRY	X	X		X	X	X	X	X	X		X		X		X			X			X	X		X
EXTERNAL		X	X				X			X		X		X						X		X		
FIXED				X		X			X		X				X				X		X			
GENERATED		X	X				X			X		X	X			X	X			X		X	X	
INITIAL		X	X						X			X	X		X		X		X	X		X		
INTERNAL							X			X				X			2							
LABEL	X	X		X	X	X	X	X	X		X		X		X			X	X		X	X		X
LOCAL		X	X				X					X				X	X			X		X		
NONLOCAL		X	X				X					X	X	2		X	X			X		X		
NORMAL	X						X		X						X			X	X					
OPTIONS	X	X		X	X	X	X	X			X		X		X			X	X		X	X		X
Parameter		X	X				X			X		X	X			X	X			X		X	X	
POINTER				X		X			X		X				X				X		X			
REGISTER		X	X	X	X	X	X	X	X	X		X	X		X	X	X		X	X		X	X	X
STATIC		X	X				X					X								X		X	X	
Structure									X						X				X			X		X

1. A component may not have a dimension attribute if a containing structure is dimensioned.

2. The combination NONLOCAL INTERNAL will be recognized as equivalent to GENERATED, but this combination should no longer be used.

INDEX

Absolute Storage Locations 25
Absolute Value 56
Address References 25,56
Addressability 40,59
Arguments 35,37,39,63
Arithmetic Data 15
Arithmetic Operations 33
Arrays 18,20
Assembler Text Inclusion 40
Assignment Statement 11,31,33,34,35,36
Associating Operators and Operands 32
Attributes 14,29,67
Automatic Storage Allocation 23

Bit Strings 15,55
Blanks 9
Block Generate 40
Boundary Alignment 26,30
Builtin Functions 25,33,56

Character Set 8
Character Strings 15,55
Code Addressability 40,59
Comments 9,10,47
Comparisons 32,34,35,41,52,55
Compile Time 2,45,53
Component 18,19,20,57,62
Composite Delimiters 8,10
Concatenation 49
Conditional Execution 12,41,51
Constants 15,37
Control Variable 35,38

Data Addressability 40,59
Data Organization 18
Data Representation 14
Data Types 15,29
Declarations 14,29,38
Default Attributes 14,29
Dimension 18,20

Entry Names 11,17,20,39,42
Epilog Activity 43,44,58,59
Expressions 25,31,35
External Procedures 20

Factored Attributes 14

Generated Data 23,40
Groups 12,42

Hexadecimal Strings 15

Identifiers 9,10
Implicit Declaration 29
Including Assembler Text 40
Indirect Addressing 16,17,24,57
Initialization 27,30
Input Line 10,40,45
Internal Procedures 13,20,43
Iterative Execution 12,38,41

Keywords 9,65
Known Data 20,40

Labels 11,17,20
Level Numbers 19
Locating Expressions 25

Macro Activation 47,52
Macro Statements 45,48
Macro Variables 46,48
Mixed Types 35,52
Multiple Closure 39
Multiple Descriptions of an Area 19,25
Multiple Initial Values 27

Name Placeholder 62
Nested Groups 12
Nested Procedures 13
Normality 28,30
Null Statement 11,42

Operations 31,32
Operators 31,32
Operator Priority 32
Options 42,43,59,63
Overlapped Data 28

Parameterization 45
Parameters 14,24,30,37,39,63
Pointer Association 16,24,57
Pointer Data 16
Pointer Qualification 16,24,57
Precision 15,16,29
Procedure Options 42,43,59
Procedures 13,20,39,42
Program Data 17
Prolog Activity 43,58,59

INDEX (continued)

Reentrant Code 23,59,60
Registers 24,43,44,58,60
Relational Expressions 32,41,51
Replication Factor 27
Rescanning 47

Save Areas 58,60,61
Scan Control 50
Scope of Names 20,29
Source Input 10,40
Statements 11,36
Static Storage Allocation 22
Storage Boundaries 26,30
Storage Class 22
String Data 15,34,55
String Operations 34
Structures 18,20,57,62
Subscripts 18,20,54
Substrings 54,55

Text Inclusion 51
Text Replacement 46,47
Transfer of Control 31,41,50
Truncation 33,34

User Generated Data 23,40

Value Assignment 31,33,49
Variable Length Substrings 55
Variable Parameter Lists 63