IBM

Basic Systems Language Primer

This publication is an introduction to the Basic Systems Language (BSL), a programming language designed for use by system programmers. The topics covered include:

- The Structure of BSL Statements

- Describing Data

- Gaining Access to Data

- Dividing Programs Into Parts

- Controlling Program Flow

- Using Assembler Language Instructions and Control Program Services

IBM Confidential

PREFACE

The purpose of this publication is to acquaint the reader with BSL, and to provide him with a general understanding of the language. It is assumed that readers have had some training or equivalent experience in programming with an assembler language.

This publication is designed to be read from beginning to end, as each chapter is a prerequisite for the following chapter. The topics covered include:

- The Structure of BSL Statements

- Describing Data

- Gaining Access to Data

- Dividing Programs into Parts

- Controlling Program Flow

- Using Assembler Language Instructions and Control Program Services

Each of the first five chapters contains a sample problem and a proposed solution. Readers are urged to complete these problems.

Additional information about BSL can be obtained from the following publications:

- BSL: Basic Systems Language Description

- BSL User's Guide

No computer could be put to use unless there were some language by which it could be directed to act. Furthermore, no computer would really be useful unless that language were meaningful not only to the computer, but also to the individual expected to use the language. Individuals have varying needs, however. These range from solving engineering problems to producing business reports; from problems requiring thousands of computations to those requiring few computations.

Whatever the need, it is doubtful whether an individual could, at least without great difficulty, direct a computer with its own language--the series of 0's and 1's that to the computer represent instructions and data.

Therefore, many programming languages have been developed. Some are general purpose languages designed for a large audience, while others are directed toward a more limited audience, but all are designed to improve the vital man-to-computer communication.

The Basic Systems Language (BSL) is a programming language designed for system programmers. It contains many features of other programming languages but, in addition, contains a number of new features particularly useful to programmers involved in developing and coding system programs.

BSL statements are English-like and are relatively easy to learn and to use. There are fourteen statements in the language. With BSL statements, programmers can describe operations to be performed without having to write intricate step-by-step instructions needed to perform the operations. For example, a decision might be described as follows:

```
IF TEST=0 THEN A=B+C; ELSE A=B-C;
```

The entire operation is described in a straightforward way, and its intent is understandable even by the untrained programmer. Once an action to be performed is clearly established, it can be described more concisely with BSL than would be possible with an assembler language.

BSL provides convenient ways for describing operations that are performed frequently in system programs. For example, tables are commonly used, and much of the work performed includes storing information in tables and retrieving the information. Access to specific items of information usually requires a complex sequence of instructions, especially when several tables are arranged in a chained list. With BSL, programmers can describe the arrangement of tables in a chain, and then use a special notation to gain access to specific items, or to add or delete tables from the chain.

BSL allows programmers to work at any level of detail required. There are times when details are extremely important, especially in system programs, where characteristics of the computer must often be considered. A control program, for example,

in which the contents of a program status word could not be examined or changed would be of little value, as would one in which main storage space could not be acquired when needed. No programming language approaches the assembler language in terms of machine-level control. Therefore, BSL allows programmers to include assembler language instructions in their programs. BSL also allows programmers to include requests for control program services in their programs.

As mentioned earlier, BSL allows programmers to describe operations to be performed without requiring them to write the instructions necessary to perform the operations. The working language of computers is machine language, however. All programs, regardless of what source language they are written in, must be converted to machine language before they can be executed. With BSL, this conversion is a two-step process involving first the BSL compiler and then the System/360 assembler.

During the first step, compilation, BSL statements are translated into the assembler language instructions required to perform the indicated operations (most BSL statements result in a number of assembler language instructions). During compilation BSL statements are checked for syntactical errors and, depending upon the severity of the error, the next step, assembly, may not be allowed to occur. The end result of compilation is a program expressed in assembler language, suitable input to a System/360 assembler. During the second step, assembly, the System/360 assembler is used to translate the assembler language instructions produced by the BSL compiler into executable machine language instructions.

BSL is a problem-oriented language, in that the notation used to describe what is to be done is more closely related to the characteristics of the application, or problem, than to those of the machine. Therefore, there is little similarity between individual BSL statements; essentially the application, more than a set of rules, governs the content of any given statement. Nevertheless, as in any language, there are rules to be followed and there are certain concepts that could be applied to most statements.

BSL statements may be written in a free-form format. That is, a statement need not begin in any particular column of a coding sheet, and more than one statement may be written on a single line. The only requirement is that the end of each statement be indicated with a semicolon.

Every BSL statement contains some symbol that denotes the operation to be performed. For the assignment statement, this symbol is an equal sign, which in that context is called an assignment symbol. For example, the statement

```
A = B;
```

is an assignment statement that reads "assign the value of B to A." For every other BSL statement, the symbol is a keyword. For example, the keyword DECLARE denotes the DECLARE statement, one used to declare, or describe, the characteristics of data items.

In general, blanks may be used freely within statements to make them more readable. The assignment statement shown above could just as well have been written in any of the following ways:

```
A=B;
A= B;
A =B;
A  = B;
A = B ;
```

The few restrictions that do exist regarding the use of blanks in BSL statements are pointed out in the text of this publication.

Programmers may use comments to clarify the meaning of any statement or group of statements. Comments must be preceded by the characters "/*" and must be followed by the characters "*/". An example of a comment is:

```
A = B; /* ASSIGN CONDITION CODE TO A */
```

In the above, the comment appears after the statement. Comments may also appear on separate lines of a coding sheet:

```
/* ASSIGN CONDITION CODE TO A */
      A = B;
```

In fact, comments may be placed anywhere that blanks are permitted:

```
A /* ENTRY IN TABLE */ = B /* CONDITION CODE */;
```

All characters used in BSL statements are selected from a BSL character set. The characters may then be used singly or in combination to form symbolic names; the symbolic names, in turn, can be used in expressions; and expressions are the basic components of most statements (Figure 1).
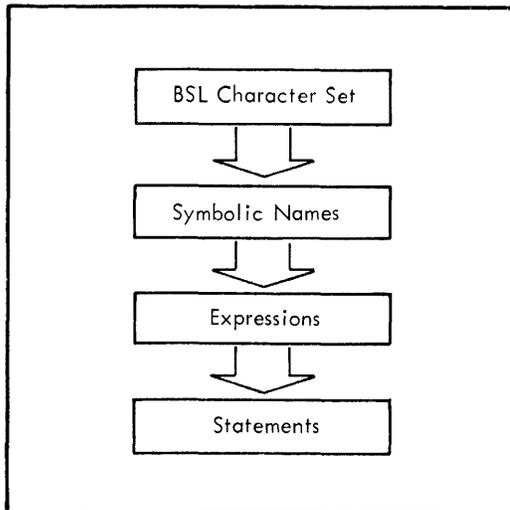

Figure 1. BSL Statement Make-up

THE BSL CHARACTER SET

The BSL character set includes the 26 letters A through Z, the 10 digits 0 through 9, and the 17 characters shown below.

NAME	CHARACTER
Blank	
"Less than" sign	<
Left parenthesis	(
Plus sign	+
"Or" sign	|
"And" sign	&
Asterisk or multiplication sign	*
Right parenthesis	)
Semicolon	;
"Not" sign	¬
Minus sign	–
Slash or division sign	/
Comma	,
"Greater than" sign	>
Colon	:
Single quotation mark or apostrophe	'
Equal sign or assignment symbol	=

Some characters may be combined to form composite symbols. For example, ¬= means "not equal to," while >= means "greater than or equal to." Blanks must not appear between such composite symbols.

The 53-character BSL character set is a subset of the EBCDIC character set, which contains 256 characters. Programmer's comments may contain any EBCDIC character, as can data being operated upon. In all other cases, however, only characters from the BSL character set may be used.

SYMBOLIC NAMES

In the assignment statement

```
TOTAL = ITEMA + ITEMB;
```

the names TOTAL, ITEMA, and ITEMB are all symbolic names. Such names must be 8 or fewer characters in length, must begin with an alphabetic character, may contain letters or numbers, but must not contain any blanks or other special characters. In BSL, as with most programming languages, such symbolic names actually represent locations. Therefore, in the assignment statement shown above, what is meant is that the sum of the values at locations ITEMA and ITEMB is to be assigned, or placed into, location TOTAL.

Symbolic names may also be given to statements to which control is to be passed. In forming statement names, the same rules as those for forming data names must be followed. In addition, a statement name must precede the statement and be separated from it by a colon, as in the following example, where CALC is the statement name.

```
CALC:  TOTAL = ITEMA + ITEMB;
```

EXPRESSIONS

When a symbolic name is used in a BSL statement, the value at the location it represents may or may not play a part in the operation being performed. For example, when ITEMA + ITEMB is written, what is meant is that the value at location ITEMA is to be added to the value at location ITEMB. The values at the locations are involved in the operation. On the other hand, when TOTAL = ITEMA + ITEMB is written the value at location TOTAL is not significant to the operation; whatever it is, it is replaced by the sum of the values at ITEMA and ITEMB -- the location TOTAL is significant, rather than the value at that location. When names are used in BSL statements in such a way that the values at the corresponding locations are significant to the operation being performed, the names are said to constitute an expression. In assignment statements, a value is always placed into a named location. Therefore, the format of the assignment statement is always:

```
symbolic name = expression;
```

An expression may consist of a single name or of a combination of names and operators. Thus, in the assignment statement

```
TOTAL = ITEMA;
```

the name ITEMA is an expression. Similarly, in the assignment statement

```
TOTAL = ITEMA + ITEMB;
```

the combination ITEMA + ITEMB is an expression. One characteristic of expressions involving two or more items of data is that they may always be reduced to a single value. Thus, in the above example, the single value produced as a result of the computation will be assigned to, or placed into, location TOTAL.

To repeat, in the above example the names TOTAL, ITEMA, and ITEMB all represent locations. Because the data at those locations can vary, the names are said to represent variables. However, if the value of ITEMA were always, say, 3, then that value could be specified directly in the statement as a constant, as follows:

```
TOTAL = 3 + ITEMB;
```

Because a constant is a symbol for a value, a single constant is always an expression.

An expression must always be part of a statement; it can never be used alone. Within statements, expressions are used to specify arithmetic, logical, and comparison

operations. Arithmetic and logical operations may be specified in most statements, including assignment statements; therefore they are described below. However, comparison operations may be specified only in the IF statement, and they are described in Chapter 5: Controlling Program Flow.

ARITHMETIC EXPRESSIONS

An arithmetic expression is one that involves addition, subtraction, multiplication, or division. The corresponding operators are the plus sign (+), the minus sign (-), the multiplication sign (*), and the division sign (/). Examples of arithmetic expressions, as used in assignment statements, are the following:

A = B+C;	C is added to B.
A = B-C;	C is subtracted from B.
A = B*C;	B is multiplied by C.
A = B/C;	B is divided by C.

The values of variables used in the expressions do not change when the arithmetic operations are performed. In all of the above, the values of B and C are unaffected by the operation. The value of A does change, however, because of the nature of the assignment statement; A will contain the results of the appropriate arithmetic operation.

When division is performed, there is usually a remainder. However, when the statement

A = B/C;

is executed, only the quotient will be assigned to A. The remainder could be obtained by using the composite operator "//", as in the following example:

A = B//C;

Two statements are necessary, then, to obtain both the quotient and the remainder when division is performed:

A = B/C; /* QUOTIENT TO A */
D = B//C; /* REMAINDER TO D */

More than one arithmetic operation may be specified with a single arithmetic expression. For example, in the statement

A = B+C*D;

the value of B will be added to the product of the values of C and D. Again, because the expression appears in an assignment statement, the result will be assigned to A.

Both the plus sign and the minus sign can be used to indicate that a particular constant or variable is to be treated as having a positive or negative value. For example, in the statement

```
A = -3;
```

the negative quantity -3 will be assigned to A. When used in this way, the plus and minus signs are called prefix operators. Another example of a prefix operator is:

```
A = -B;
```

In the above, the sign given to A will depend upon that of B, as prefix operators are applied algebraically. That is, if B is a negative quantity, A will be positive; if B is a positive quantity, A will be negative. The sign of a variable does not change when a prefix operator is specified for it; B is not changed when the assignment shown above is performed.

There is a specific order by which complex expressions are evaluated. First, prefix operators are applied. Next, multiplication and division are performed. Addition and subtraction are performed last. These priorities are shown in Figure 2. When two or more operations of the same priority appear in the same expression, they are performed in left-to-right order. For example, in the statement

```
A = B+C/D-E*F;
```

the order of evaluation is:

1. C is divided by D.
2. E is multiplied by F.
3. B is added to the quotient of C/D.
4. The product of E*F is subtracted from the result of B+C/D.

Parentheses may be used to group items of an arithmetic expression either to make the meaning of the expression more evident, or to override the priority by which the operations are performed. The same result would be obtained if the previous example were written as follows:

```
A = (B+C/D)-(E*F);
```

However, a different result would be obtained if the statement were written as follows:

```
A = B+(C/D-E)*F;
```

Operator	Operation	Priority
+	Prefix Plus	1
-	Prefix Minus	1
*	Multiplication	2
/	Division (Quotient)	2
//	Division (Remainder)	2
+	Addition	3
-	Subtraction	3

Figure 2. Arithmetic Operators

LOGICAL EXPRESSIONS

Logical expressions provide a convenient and efficient means for selectively altering the bit settings of data items. With logical expressions, any of three Boolean logical operations can be specified--and, or, and exclusive or.

The result of a logical expression always depends upon two factors: the operation specified, and the settings of the bits in two data items. A logical operation can be specified with the operators & (and), | (or), and && (exclusive or); the two data items may be constants or variables. The corresponding bits of the two data items are examined, bit by bit, and, depending upon the operation specified and the settings of each pair of bits being examined, the value of the expression is developed. This value may then be assigned to a variable, as with an assignment statement, or it may be used as an operand in an arithmetic operation. The table below shows the results of bit-pair combinations for each logical operation.

A	B	A&B	A|B	A&&B
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

With an 'and' operation, when the corresponding pair of bits in two data items are both 1, the resulting bit will be set to 1; otherwise, it will be set to 0. An 'and' operation involving two data items, ITEMA and ITEMB, would be specified as follows:

ITEMA & ITEMB

The result, or value of the expression, would be developed as follows:

ITEMA	1	0	1	0	1	0	1	0
ITEMB	1	1	1	1	0	0	0	0
ITEMA & ITEMB	1	0	1	0	0	0	0	0
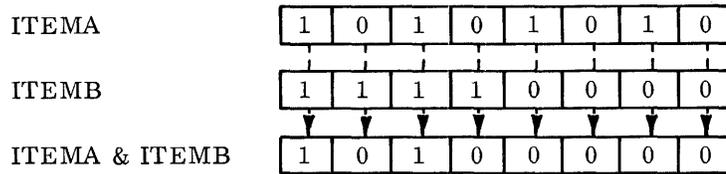
If the expression shown above appeared in the assignment statement
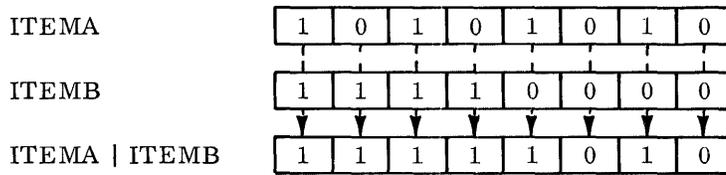
    ITEMC = ITEMA & ITEMB;

the result would be assigned to ITEMC. The values of ITEMA and ITEMB would not be changed. If it were desired that, say, ITEMB were to contain the result, the following assignment statement could have been written:

    ITEMB = ITEMA & ITEMB;

With an 'or' operation, when one or the other of the corresponding pair of bits in two data items is 1, the resulting bit will be set to 1; otherwise it will be set to 0. An 'or' operation involving the two data items, ITEMA and ITEMB, would be specified as follows:

    ITEMA | ITEMB

The value of the expression would then be developed as follows:

| ITEMA	1	0	1	0	1	0	1	0
ITEMB	1	1	1	1	0	0	0	0
ITEMA | ITEMB	1	1	1	1	1	0	1	0

The expression shown could be specified in an assignment statement as follows:

    ITEMC = ITEMA | ITEMB;

With an 'exclusive or' operation, when one but not both of the corresponding pair of bits in two data items is 1, the resulting bit will be set to 1; otherwise it will be set to 0. Using ITEMA and ITEMB again, an 'exclusive or' operation could be specified as follows:

    ITEMA && ITEMB

The value of the expression would be developed as follows:

ITEMA		1	0	1	0	1	0	1	0
ITEMB		1	1	1	1	0	0	0	0
ITEMA && ITEMB		0	1	0	1	1	0	1	0
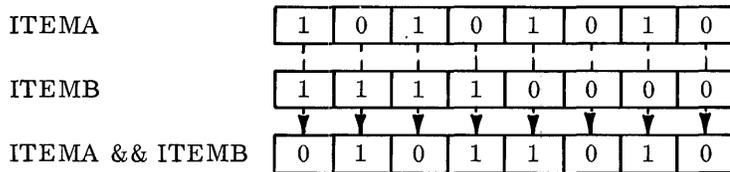
The expression could appear in an assignment statement as follows:

```
ITEMC = ITEMA && ITEMB;
```

There are many applications for which logical expressions can be used, including setting and testing switches, and setting portions of an item to 0 (an entire item may be set to 0 if it is 'exclusive or'ed' to itself). Logical expressions should be considered whenever bits are to be set "on" or "off."

When logical operations are specified in complex expressions, the priorities shown in Figure 3 are applied.

Operator	Operation	Priority
&	And	5
I	Or	6
&&	Exclusive Or	7

Figure 3. Logical Operators

CHECKPOINT 1: PROBLEM

Select any three values, assign them to the variables A, B, and C, and then:

1. Add the values together.
2. Multiply the sum of the values by each value to obtain three products.
3. Add the three products obtained from (2) above.

A solution to this problem appears on the following page.

```
/* SOLUTION TO CHECKPOINT 1 PROBLEM */

/* CALCULATE SUM OF THREE VALUES */
        A = 2;
        B = 4;
        C = 6;
        SUM = A + B + C;

/* MULTIPLY SUM BY EACH VALUE */
        PRODUCT1 = SUM * A;
        PRODUCT2 = SUM * B;
        PRODUCT3 = SUM * C;

/* ADD PRODUCTS TOGETHER */
        SUMPROD = PRODUCT1 + PRODUCT2 + PRODUCT3;
```

With an assembler language, there are usually several instructions that might be used to perform a particular operation. For example, if the operation is one in which two items are to be added together, an Add instruction would be used when fullword quantities are involved and the number is signed; an Add Logical instruction would be used when all bits are significant; or, when halfword quantities are involved, an Add Halfword instruction would be used. It can be said, then, that with an assembler language, the characteristics of the data play a part in the selection of instructions to be used in a program.

The same characteristics must also be considered when using BSL. However, the characteristics are not reflected in BSL statements as they are in assembler language instructions. To illustrate, it is not evident from the following statement exactly what the characteristics of the items are:

> TOTAL = ITEMA + ITEMB;

Therefore, the operation shown could not be performed unless additional information were provided to the BSL compiler, which could then select the appropriate instructions to be used in acting upon the data.

The characteristics, or attributes of data items can be made known to the compiler with a statement called the DECLARE statement. The DECLARE statement is the primary means by which data can be described. It consists of the statement keyword DECLARE, followed by the name to be given a location, or area of storage, in turn followed by the attributes of the item that is to occupy the area:

> DECLARE   name   attributes;

The attributes of a data item can be expressed in terms of an attribute keyword and a length. The attribute keyword indicates in what way the bit configuration of an item is to be interpreted, while the length indicates how much space the item is to occupy. Once an item is declared, should it ever become necessary to change the attributes of the item, only the DECLARE statement for that item need be changed; individual statements in which the item is referred to would require no change whatsoever.

There are four ways in which the bit configuration of an item can be interpreted, or, in other words, there are four types of data.

TYPES OF DATA

The four types of data are arithmetic data, string data, pointer data, and program data. To review, the type indicates in what way the bit configuration of an item is to be interpreted.
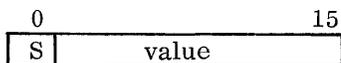
- Arithmetic data is data for which the bit configuration is taken to be a signed numeric quantity.

- String data is data for which the bit configuration is taken to be a sequence or string of 8-bit EBCDIC characters, or else a sequence of bits that are unrelated to each other (as when individual bits serve as switches).

- Pointer data is data for which the bit configuration is taken to be an address of another data item.

- Program data, like pointer data, is taken to be an address. However, it is always taken to be an address of an instruction.

Each type of data can be described with a DECLARE statement containing the item name and the attributes to be associated with the item. With this information, the BSL compiler can select the instructions to be used in manipulating the data in light of its intended interpretation.

Both variables and constants can be classified by type. However, when a constant is used, it does not have to be described with a DECLARE statement. Instead, there are different ways of expressing constants, and the way a constant is expressed determines how it is to be treated by the compiler.

ARITHMETIC DATA

An area of storage containing an item of arithmetic data is always assumed to contain a signed, fixed-point number. Internally, there is a specific format by which such numbers are represented. A fixed-point number can be contained in a halfword or a fullword, the sign position is always the high-order bit of the item, and the value of the item occupies the remaining bits, as follows:

```
0                    15
┌─┬──────────────────┐
│S│     value        │
└─┴──────────────────┘

0                              31
┌─┬────────────────────────────┐
│S│     value                  │
└─┴────────────────────────────┘
```

The attribute keyword to be used in the DECLARE statement for an arithmetic item is FIXED. The specified length of the item, which should follow the FIXED keyword, can be either 15 or 31, depending upon whether the maximum value can be represented with 15 or 31 bits. To illustrate, if ITEMA is to contain a quantity or, during the course of execution, several quantities, but none will require more than 15 bits, the declaration could be:

```
DECLARE ITEMA   FIXED (15);
```

Should ITEMA require 31 bits, the declaration would be:

```
DECLARE ITEMA FIXED (31);
```

It is not always necessary to declare an item to be an arithmetic item. Any time a symbolic data name is used in a statement, and the name is not declared, the item is automatically taken to be a fullword arithmetic quantity. In other words, the attributes FIXED (31) are applied by <u>default</u> to any undeclared data variables.

There are times when the context of a statement indicates that the item or items involved are to be treated as arithmetic quantities, regardless of how they have been declared. For example, whenever an arithmetic operator (+, -, *, /, or //) is contained in an expression, the items involved are all treated as arithmetic quantities. If such an expression appears in an assignment statement, all items referred to by the statement are treated as arithmetic quantities. For example, when the statement

```
TOTAL = ITEMA + ITEMB;
```

is written, even if TOTAL, ITEMA, and ITEMB are declared to be a type other than arithmetic, they will be assumed to be arithmetic quantities, as the arithmetic operator (+) is used.

Also, if the symbolic name to the left of the equal sign in an assignment statement has been declared to represent an arithmetic item, any other items referred to by the assignment statement will be taken to be arithmetic items, regardless of how they have been declared.

When an arithmetic constant is to be used in a statement, it may be expressed in either of two ways. It can be expressed as a decimal number or as a binary number that is followed by the letter B. The two statements below will each result in the same internal representation of the quantity 53:

```
TOTAL = 53;
```

```
TOTAL = 110101B;
```

Constants are commonly used to assign initial values to variables. For example, when the assignment statement shown below is executed, ITEMA will be given an initial value of 350:

```
DECLARE ITEMA FIXED (15);
ITEMA = 350;
```

The same result could be obtained by including an INITIAL attribute keyword in the declaration for ITEMA, and following that keyword with a constant, in parentheses, which indicates what the initial value of the variable is to be:

```
DECLARE  ITEMA  FIXED  (15)  INITIAL  (350);
```

or

```
DECLARE  ITEMA  FIXED  (15)  INITIAL (101011110B);
```

A number of items can be described with the same DECLARE statement, if the names and attributes of each are separated by commas:

```
DECLARE  ITEMA  FIXED  (15),  ITEMB  FIXED  (15),
         ITEMC  FIXED  (15);
```

In addition, attributes common to more than one data item may be factored by enclosing the names of the items in parentheses, and specifying any common attributes following the closing parenthesis:

```
DECLARE  (ITEMA,  ITEMB,  ITEMC)  FIXED  (15);
```

STRING DATA

It has been shown how, by declaring an item to be FIXED or by not declaring the item at all, a programmer could cause the representation of the item to be interpreted as being a numeric quantity. If another interpretation were intended, then an attribute keyword other than FIXED would have to be specified in the DECLARE statement.

EBCDIC characters, for example, belong to a category of data called string data, of which there are two sub-categories, each of which has an associated attribute keyword. An item can be termed a character string; that is, a sequence of bits that are to be interpreted as representing EBCDIC characters, or an item can be termed a bit string; that is, a sequence of bits that may have no meaning collectively, but do have significance individually, as when they are being used as switches.

Character Strings

An area of storage declared to contain a character string will be interpreted as containing a representation of EBCDIC characters. Internally, one byte is required to represent each character, and each character has a specific bit configuration. For example, the character string AB12 would be represented internally as follows:

11000001	11000010	11110001	11110010

The attribute keyword to be used in the DECLARE statement to identify an item as being a character string is CHARACTER. The keyword must be followed, in parentheses, by the number of characters in the item. For example, the four-character area shown above could be given the name ITEMA and declared to be a character string with the following DECLARE statement:

```
DECLARE  ITEMA  CHARACTER  (4);
```

If ITEMA is to be used in an assignment statement, another area, the one to which ITEMA is to be assigned, would also have to be declared. For example, if the area were to be 6 characters in length, the following DECLARE statement might be written:

```
DECLARE  ITEMB  CHARACTER  (6);
```

Now, the following assignment statement can be written:

```
ITEMB = ITEMA;
```

After the assignment is performed, the first four bytes of ITEMB will contain a representation of the characters AB12, while the next two bytes will contain the EBCDIC representation of blanks. Character string assignment is always performed from left to right. If a character string is assigned to an area of storage that consists of more bytes than there are characters, as was the case in the preceding example, surplus bytes will be set to a representation of EBCDIC blanks. However, if a character string is assigned to an area containing fewer bytes than there are characters in the string, the right-hand portion of the string will be truncated.

The statements used in the previous examples might appear in a program as follows:

```
DECLARE  ITEMA  CHARACTER  (4);
DECLARE  ITEMB  CHARACTER  (6);
         .
         .
         .
ITEMB = ITEMA;
```

As with arithmetic data, character strings may be expressed as constants. When a character string constant is used, it must be enclosed in single quotation marks to distinguish it from a symbolic name or from other types of constants. Assume that the character string SUBTOTAL is to be used as a heading in a report to identify a particular column, and that, before it can be printed, it is to be placed into an area named HEAD1. The declaration and assignment statement might be:

```
DECLARE HEAD1 CHARACTER (8);
HEAD1 = 'SUBTOTAL';
```

Because a blank is a valid EBCDIC character, blanks should not appear between the single quotation marks and the first or last character in the character string unless it is intended that they be a part of the string. Also, note that if the single quotation marks were omitted from the character string constant SUBTOTAL, it would be taken to be a symbolic name, rather than a constant.

A hexadecimal constant may be used to assign a value to a data item declared to be a character string. Hexadecimal constants must be enclosed in single quotation marks and followed by the letter X. To illustrate, the assignment statement shown below will cause a one-byte area, ITEMA, to be set to ones.

```
DECLARE ITEMA CHARACTER (1);
ITEMA = 'FF'X;
```

Bit Strings

Often, it is desirable to use a single bit for some purpose. For example, a bit could serve as a switch to be set "on" or "off" when certain conditions arise, and its setting could later be used in selecting a particular action to be taken. A number of contiguous bits might be used, and each could be a separate switch. When used in this way, the bits do not collectively represent a numeric quantity or an EBCDIC character but, rather, serve individually. When individual bits of an item are to have significance, the item should be declared to be a bit string. (Although the term bit string implies two or more bits, a single bit can be declared to be a bit string.)

The attribute keyword needed in the DECLARE statement when a bit string is to be described is BIT. The number of bits in the string should be placed in parentheses following the keyword BIT. To illustrate, assume that ITEMA is to be a bit string, and that it is to contain four bits. The declaration would be:

```
DECLARE ITEMA BIT (4);
```

A bit string constant could then be used to "set" the four bits in ITEMA. As a convenience, any bit string constant can be expressed in either of two ways: in binary notation, enclosed by single quotation marks and followed by the letter B; or in hexadecimal notation, also enclosed in single quotation marks, but followed by the letter X. The two statements shown below will each result in the same bit configuration for ITEMA:

```
ITEMA = '1011'B;  /*  BINARY NOTATION  */
ITEMA = 'B'X;     /*  HEXADECIMAL NOTATION  */
```

POINTER DATA

An area of storage declared to contain a pointer data item will be interpreted as containing a representation of an address. Pointer data items are normally used when it is necessary or desirable to use a technique called indirect addressing. With indirect addressing, an item of data is obtained indirectly; that is, via another data item containing the address of the desired item. The way in which indirect addressing can be achieved is described in Chapter 3: Gaining Access to Data.

The attribute keyword to be used in the DECLARE statement when a pointer data item is to be described is POINTER. The keyword should be followed by a number, in parentheses, indicating the number of bits needed to represent the address that is to occupy the area. This number can be 8, 15, 16, 24, 31, or 32. The number of bytes set aside for an item when these lengths are specified is shown below:

No. of Bits Declared	No. of Bytes Set Aside
8	1
15	2
16	2
24	3
31	4
32	4

Any item declared to be a pointer data item may be used in arithmetic operations. However, pointer data items are always taken to be positive integers. Thus, every bit in such items has significance.

Assume that location ITEMA is to contain an address, and that the maximum value of the address can be expressed in 24 bits. The declaration could be either of the following:

```
DECLARE ITEMA POINTER (24);
```

or

```
DECLARE ITEMA POINTER (31);
```

An address can now be placed into the area of storage named ITEMA by making use of a BSL "Built-in Function" called the ADDR function. Basically, when an expression is made up of the keyword ADDR which is followed by a symbolic name, in parentheses, the value of the expression is taken to be the address of the location represented by the symbolic name. For example, when the expression

```
ADDR (ITEMB)
```

is written, its value will be taken to be the address of ITEMB. Using the ADDR function, the address of ITEMB can be assigned, or placed into location ITEMA as follows:

```
ITEMA = ADDR (ITEMB);
```

Thus, to declare an item that is to contain an address, and to assign an address to the item, the following statements could be used:

```
DECLARE ITEMA POINTER (24);
ITEMA = ADDR (ITEMB);
```

PROGRAM DATA

When a symbolic name is used as a statement label, it is converted by the compiler to an address -- that of an instruction (or the first of a sequence of instructions) to be used in performing the operation designated in the statement. Such addresses belong to a catagory of data called program data. There are two attribute keywords that can be used in a DECLARE statement to identify program data items: ENTRY and LABEL. Both are default attributes. The ENTRY attribute is applied automatically to entry points for procedures (procedures are described in Chapter 4: Dividing Programs Into Parts). The LABEL attribute is applied automatically to statement names.

When the address of a label or of an entry point is desired, it can be obtained by using the LABEL or ENTRY attributes and also the ADDR function:

```
DECLARE A LABEL;
DECLARE ADDRA POINTER INITIAL (ADDR (A));
    .
    .
    .
A: TOTAL = ITEMA + ITEMB;
```

The above declarations will result in a data item, ADDRA, which will contain the address of the instruction or the first of the sequence of instructions that are produced by the compiler for the statement named A.

COLLECTIONS OF DATA

So far in this chapter, each data name in the discussions and examples shown has represented a single item of data.

In BSL, data can be named so that an entire collection of data items can be referred to by one name; certain subcollections can be referred to be other names, and individual items can be referred to separately.

There are two kinds of data collections to which collective names can be given: structures and arrays. The major difference between the two is in the kind of data an array name or a structure name represents. An array is a table of homogeneous data items; that is, all items in an array have identical attributes. In an array of arithmetic items, for example, all of the items might have the attributes FIXED (31); in a character-string array, all of the items would consist of an equal number of characters. The data items of a structure, on the other hand, need not be of the same data type nor have the same attributes. Some names in a structure might represent string data while other names in the same structure could represent arithmetic data.

Collective naming does not alter the data in any way. Data in a structure or an array is no different from what it would be if it were referred to by a single name. Collective naming merely gives a programmer more convenience in referring to and manipulating data.

STRUCTURES

A structure is a graded system of names that represent items or groups of items in a single area of internal storage. At the highest level, a single name--the major structure name--represents all data items stored in the entire area that is allocated to that structure. At the next level, certain portions of the area are individually named; at the deepest level, each name is a variable.

To illustrate, assume that a program is to perform operations upon the following collection of data items:
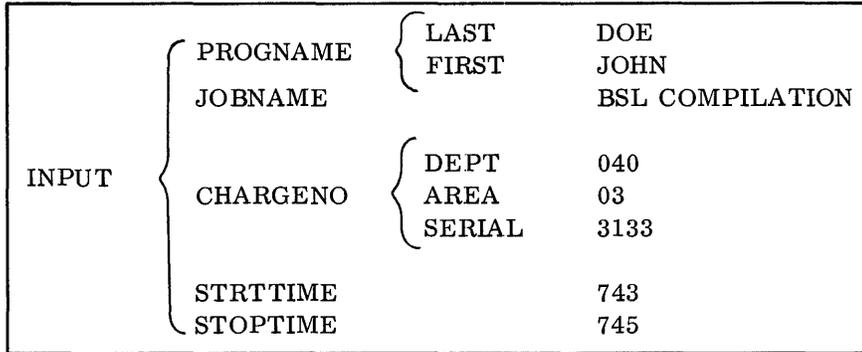
Programmer's Name	JOHN DOE
Name of Job	BSL COMPILATION
Charge No.	040 03 3133
Time Job Started	743
Time Job Completed	745

If the collection were represented by a single name, say INPUT, it would be difficult to refer to individual items. However, a symbolic name could also be given to each item:

	PROGNAME	JOHN DOE
	JOBNAME	BSL COMPILATION
INPUT	CHARGENO	040 03 3133
	STRTTIME	743
	STOPTIME	745

Now, a programmer can refer to the entire collection of data items by the name INPUT, or he can refer to an individual item by its individual name.

It often is valuable to be able to refer collectively to more than one, but not all, of the variables in a structure. If such is the case, an intermediate structure can be built:

```
            ⎧           ⎧ LAST        DOE
            ⎪ PROGNAME  ⎨ FIRST       JOHN
            ⎪           ⎩
            ⎪ JOBNAME                 BSL COMPILATION
            ⎪
            ⎪           ⎧ DEPT        040
INPUT   ⎨  CHARGENO   ⎨ AREA        03
            ⎪           ⎩ SERIAL      3133
            ⎪
            ⎪ STRTTIME                743
            ⎩ STOPTIME                745
```

The major structure, INPUT, contains the structures PROGNAME and CHARGENO. JOBNAME, STRTTIME, and STOPTIME are not themselves structures because each represents only a single data item.

A reference to any structure is a reference to all the data items included in that structure. In other words, a reference to any name is really a reference to the names at the next deeper level. To illustrate, a reference to the name at the first level (INPUT, in the example) is a reference to all of the names at the second level (PROGNAME, JOB-NAME, CHARGENO, STRTTIME, and STOPTIME). A reference to a name at the second level (CHARGENO) is a reference to all of the names in that structure at the third level (DEPT, AREA, and SERIAL).

Since there are no levels deeper than those represented by the names DEPT, AREA, and SERIAL, the corresponding items are not structures. They are elementary items, like JOBNAME.

Structures are described in the DECLARE statement. When a structure is declared, the level of each name is indicated by a level number. The major structure name, at the highest level, is always given the level number 1. Names at deeper levels are given numbers to indicate the level depth, and all names are separated by commas:

```
DECLARE    1  INPUT, 2  PROGNAME, 3  LAST, 3  FIRST,
           2  JOBNAME, 2  CHARGENO, 3  DEPT, 3  AREA,
           3  SERIAL, 2  STRTTIME, 2  STOPTIME;
```

The same DECLARE statement could be written as follows:

```
DECLARE 1 INPUT,
     2 PROGNAME,
          3 LAST,
          3 FIRST,
     2 JOBNAME,
     2 CHARGENO,
          3 DEPT,
          3 AREA,
          3 SERIAL,
     2 STRTTIME,
     2 STOPTIME;
```

The order of the appearance of the names in a DECLARE statement, along with their level numbers, determines the structuring. Except for the major structure name, no particular numbers must be specified. The major structure name must be declared with the level number 1, and it must be the first name listed in the structure declaration. It must be followed by one of the names at the second level. Each name at the second level must then have a level number greater than 1, and if it is a structure it must be followed by the names within that structure. Each structure must be completely declared before the next structure declaration begins, and each must have a level number equal to or less than the level number of the immediately preceding structure at the same level. For example, the structuring shown above could be achieved if INPUT were declared in the following way:

```
DECLARE 1 INPUT,
     8 PROGNAME,
          20 LAST,
          20 FIRST,
     6 JOBNAME,
     3 CHARGENO,
          4 DEPT,
          4 AREA,
          4 SERIAL,
     3 STRTTIME,
     3 STOPTIME;
```

When a structure is declared, attributes may be specified for individual items within the structure; any items for which attributes are not specified will be given the attributes FIXED (31) by default:

```
DECLARE 1 INPUT,
    2 PROGNAME,
        3 LAST CHARACTER (10),
        3 FIRST CHARACTER (8),
    2 JOBNAME CHARACTER (15),
    2 CHARGENO,
        3 DEPT FIXED (15),
        3 AREA FIXED (15),
        3 SERIAL FIXED (15),
    2 STRTTIME FIXED (15),
    2 STOPTIME FIXED (15);
```

The level numbers that must appear with structure names in the DECLARE statement should not appear with the names in any references to them. For example, the name John Doe could be assigned to the variables LAST and FIRST of the structure PROGNAME with the following assignment statements:

```
LAST  = 'DOE';
FIRST = 'JOHN';
```

As with any DECLARE statement, attributes common to more than one data item may be factored by enclosing the names of the items in parentheses and specifying the common attributes following the closing parenthesis:

```
DECLARE 1 INPUT,
    2 PROGNAME,
        3 LAST CHARACTER (10),
        3 FIRST CHARACTER (8),
    2 JOBNAME CHARACTER (15),
    2 CHARGENO,
        (3 DEPT, 3 AREA, 3 SERIAL) FIXED (15),
    (2 STRTTIME, 2 STOPTIME) FIXED (15);
```

Structures may be given attributes, and a structure may validly be declared in which the designated size differs from the combined lengths of its elements. When the size of a structure is smaller than the combined sizes of its elements, the elements will not be truncated. Instead, the portions of any elements that extend beyond the designated size of the structure will overlap the following structure:

DECLARE 1 EXAMPLE,
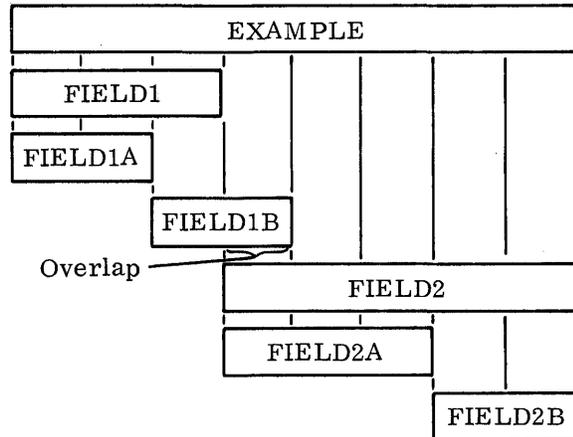
    2 FIELD1 CHARACTER (3),

      3 FIELD1A CHARACTER (2),

      3 FIELD1B CHARACTER (2),

    2 FIELD2 CHARACTER (5),

      3 FIELD2A CHARACTER (3),

      3 FIELD2B CHARACTER (2);

ARRAYS

An array is a named table of data items all of which have identical attributes. Only the array is given a name. An individual item of an array is referred to by giving its location within the array. The location is specified by a <u>subscript</u> following the array name.

Assume TABLE has been declared to be an array of 12 elements. TABLE (1) refers to the first item in the list, TABLE (2) to the second, TABLE (3) to the third, etc. Each of the numbers (1), (2), or (3), is a subscript that gives the location, within TABLE, of a particular data item.

An array is described in a DECLARE statement by giving its name, the number of elements in the array, and the attributes of the items:

```
DECLARE TABLE (12) FIXED (15);
```

The above declaration specifies that the name TABLE refers to an array of 12 data items, each of which will have a value that can be represented in a halfword. TABLE is declared to have the <u>dimension attribute</u> of (12). The bounds of the array are 1 and 12. In array declarations, the dimension attribute must immediately follow the array name.

Assume that the following values have been assigned to the items in TABLE:

31
43
42
57
64
73
79
79
69
58
49
40

Thus, TABLE (1) would refer to the data item 31, TABLE (6) to 73, TABLE (12) to 40. The expression TABLE (7) + TABLE (1) would yield a value of 110.

Initial values may be given to any or all items in an array by specifying the INITIAL attribute in the declaration for the array, and then by placing the desired values in parentheses. For example, the table shown previously could have been declared and initialized as follows:

```
DECLARE TABLE  (12)  FIXED  (15)
      INITIAL  (31,  43,  42,  57,  64,  73,  79,  79,  69,  58,
              49,  40);
```

An asterisk may be used in place of any initial values to indicate those items in an array that are not to be initialized. Also, any value that is to be repeated a number of times in the declaration for the array may be shown by preceding the value with a parenthesized replication number:

```
DECLARE TABLE  (12)  FIXED  (15)
      INITIAL  ((2) *, (10)55);
```

The above declaration would result in an array of 12 items, the first two of which would not be initialized, and the remaining 10 of which would each be given initial values of 55.

ARRAYS OF STRUCTURES

As discussed previously, all items in an array must have identical attributes. It follows then, that if two or more structures have identical attributes, they can be made to be elements of an array. An array of structures can be formed by adding a dimension attribute to the first level of a structure:

```
DECLARE  1  TABLE  (5),
     2  ADDNEXT  POINTER,
     2  ADDLAST  POINTER,
     2  DESC  CHARACTER  (12);
```

The above declaration would result in five elements called TABLE, each of which would contain the data items ADDNEXT, ADDLAST, and DESC. A specific structure in the array could later be referred to by using the appropriate subscript: TABLE (4) would be the fourth structure, or table, in the array. Similarly, a subscript could also be used to refer to a specific item within a particular structure. For example, ADDLAST (4) would be taken to mean the item ADDLAST of the fourth structure.

ARRAYS WITHIN STRUCTURES

Any item in a structure can be declared to be an array. For example, in the declaration

```
DECLARE  1  TABLE,
     2  ADDNEXT  POINTER,
     2  ADDLAST  POINTER,
     2  DESC  (4)  CHARACTER  (3);
```

The item DESC in the structure TABLE is an array of 4 data items, each of which is 3 characters long. In subsequent statements, DESC (1) would be taken to mean the first item of the array, DESC (2) the second item, etc. One restriction exists regarding arrays within structures: Items within a structure that is an array cannot themselves be arrays. In the above, if TABLE were declared to be an array, none of the items within TABLE could be arrays.

CHECKPOINT 2: PROBLEM

Write the statements necessary to translate the content of any one-byte area to the character codes of the two hexadecimal characters it contains. For example, if a one-byte area contains the following:

```
00101110
```

the hexadecimal equivalent would be '2E', and the character codes for the characters '2E' would be:

| 11110010 | 11000101 |

A suggested method would be to do the following:

1. Declare and initialize a one-byte area, say BYTE, with some value.

2. Declare an array of 16 elements, and initialize it with the character codes for each of the 16 hexadecimal characters:

0	11110000
1	11110001
2	11110010
3	11110011
.	.
.	.
.	.
D	11000100
E	11000101
F	11000110

3. Declare a two-byte area, say TWOCHARS, into which the two character codes for the hexadecimal characters in BYTE will be placed.

4. Obtain the first four bits of the byte to be converted, and add 1 to their decimal value. The result will reflect the position in the array of the character code for the appropriate hexadecimal character. For example, if the four bits are 0010, the result, when 1 is added, will be 0011, or 3. The character representation of the hexadecimal 0010 occupies the third position of the array.

5. Using the value obtained from step 4 as a subscript, locate the appropriate entry in the array and assign the content of that entry to the first byte of TWOCHARS.

6. Repeat steps 4 and 5, this time working with the last four bits of BYTE; assign the character code to the second byte of TWOCHARS.

A solution to this problem appears on the following page.

```
/* SOLUTION TO CHECKPOINT 2 PROBLEM */

/* INITIALIZE ONE-BYTE AREA, BYTE, WITH CHARACTER A */
        DECLARE BYTE CHARACTER (1);
        BYTE = 'A';
/* CONSTRUCT TABLE OF HEXADECIMAL EQUIVALENTS */
        DECLARE HEXREP (16) CHARACTER (1) INITIAL ('0', '1', '2', '3',
        '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F');
/* HEXADECIMAL EQUIVALENT OF BYTE WILL BE PLACED INTO TWOCHARS */
        DECLARE 1 TWOCHARS,
                2 FSTCHAR CHARACTER (1),
                2 SECCHAR CHARACTER (1);
/* FOURBITS WILL BE A WORK AREA */
        DECLARE FOURBITS CHARACTER (1);
/* CONVERT FIRST FOUR BITS OF BYTE TO HEXADECIMAL */
        FOURBITS = BYTE & 'F0'X;        /* ZERO LOW-ORDER FOUR BITS */
        FOURBITS = FOURBITS / 16;       /* SHIFT RIGHT FOUR PLACES */
        FOURBITS = FOURBITS + 1;        /* COMPENSATE FOR POS. IN TABLE */
        FSTCHAR  = HEXREP (FOURBITS);/* HEX REP. TO FSTCHAR */
/* CONVERT SECOND FOUR BITS TO HEXADECIMAL */
        FOURBITS = BYTE & '0F'X;        /* ZERO HIGH-ORDER FOUR BITS */
        FOURBITS = FOURBITS + 1;        /* COMPENSATE FOR POS. IN TABLE */
        SECCHAR  = HEXREP (FOURBITS);/* HEX REP. TO SECCHAR */
/* TWOCHARS NOW CONTAINS THE HEXADECIMAL EQUIVALENT */
/* OF THE ONE-BYTE AREA, BYTE */
```

CHAPTER 3: GAINING ACCESS TO DATA

The simplest way to gain access to an item of data is to give the item a name, and then to use that name in a statement. The manner in which this can be done has been described in earlier chapters. In some cases, however, only a part of a particular item is needed, or the needed item is to be located indirectly, via a pointer data item. In both cases, more than just a name is required; the data can be referred to by using a particular notation in the statement that is to act upon the data.

The notation required when a part of an item is needed is called substring notation. As the name implies, substring notation may be used only with string data. Essentially, when a portion of a string is needed, the bounds of that portion are indicated, in parentheses, following the name of the item.

The notation required when an item is to be located indirectly, via an address contained in another item is called pointer notation. With pointer notation the name of the item containing the address is given, followed by the composite operator ->, followed by the name of the needed item.

SUBSTRING NOTATION

Substring notation is similar to subscript notation, which has been described as the means of referring to a specific item in an array. To review, any item in an array may be referred to by giving the name of the array and following that name with a number, in parentheses, which indicates the position in the array of the needed item. With substring notation, any part of an item may be referred to by giving the name of the item and following that name with a number that indicates the position of the desired part of the item. To illustrate, assume that two items, ITEMA and ITEMB, have been declared as follows:

```
DECLARE ITEMA CHARACTER (8);
DECLARE ITEMB CHARACTER (3);
```

The fifth character of ITEMA can be assigned to the first byte of ITEMB as follows:

```
ITEMB (1) = ITEMA (5);
```

If more than one character of an item is desired, two numbers, separated by a colon, must be used. The two numbers should indicate the positions of the first and last characters of the substring. For example, if characters 4-6 of ITEMA are to be assigned to ITEMB, the assignment statement would be:

```
ITEMB = ITEMA (4:6);
```

If ITEMA and ITEMB were declared to be bit strings, and a specific bit or sequence of bits were desired, the same notation could be used. The only difference would be that the numbers in parentheses would represent bit positions rather than character positions.

If an item is contained in an array, a portion of the item can be referred to by first indicating the position of the item in the array, following it by a comma, and then indicating the position of the desired part of the item. For example, if ITEMA and ITEMB were declared as follows:

```
DECLARE ITEMA (15) CHARACTER (3);
DECLARE ITEMB CHARACTER (2);
```

The second and third characters of the last item in the array ITEMA could be assigned to ITEMB with the following statement:

```
ITEMB = ITEMA (15,2:3);
```

A variable-length substring would result if symbolic names are used to indicate the bounds of the desired part of an item. For example, the length of the substring in the following statement would depend upon the values of FIRST and LAST:

```
ITEMB = ITEMA (FIRST:LAST);
```

When symbolic names are used as above, there are certain rules that must be followed:

- Variable-length substrings may not be used as operands in arithmetic operations.

- When a data item is assigned to a variable-length substring, the length of the substring must not be greater than that of the data item.

- When a variable-length substring is assigned to an item, the length of the substring must not be greater than that of the item.

- A symbolic name may not be used to indicate a single-bit substring.

- When at least one symbolic name is used to indicate a bit substring, the first position of the substring must be the leftmost bit in a byte, and the last position must be the rightmost bit in some (possibly the same) byte.

POINTER NOTATION

When an array of structures is declared, one block of space is assigned to the entire array, and each structure in that array is positioned adjacent to the other structures.

For example, if the declaration

```
DECLARE 1 TABLE (3),
    2 ITEMA,
    2 ITEMB;
```

is written, the three structures within TABLE will be arranged in storage as follows:

TABLE (1)	ITEMA
	ITEMB
TABLE (2)	ITEMA
	ITEMB
TABLE (3)	ITEMA
	ITEMB
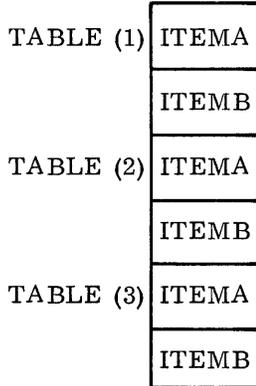
Thus, an expression such as

```
ITEMB (3)
```

is allowed because it is possible for the compiler to gain access to ITEMB of the third structure in the array. That is, the beginning address of the array is known, and the lengths of each structure are known; therefore, the displacement of ITEMB (3) from the beginning of the array could be calculated.

In practice, structures may occupy a large amount of space, or the number of structures needed may not be known beforehand. Therefore, it might be impractical to declare an entire array of structures at one time. An alternative would be to declare each structure as it is needed. When this is the case, however, it would be unlikely that the structures would be adjacent to each other. Therefore, displacements could not be used to locate a needed structure.

When an application is such that a chain of non-contiguous structures (or tables) is created, programmers commonly keep track of the locations of the tables by inserting into each the beginning address of the next table in the chain.

Pointer notation simply provides a way of indicating such an arrangement to the compiler, so it can use the addresses to gain access to specific tables. Once the beginning address of a table is known to the compiler, it can readily compute the displacement of a needed item from the beginning of that table.

Pointer notation involves the use of the composite operator –>, which is preceded by the name of a pointer variable, and followed by the name of a data item. (To review, pointer variables are always addresses by which other data items can be located.)

To begin with an uninvolved example, the expression

```
A -> B
```

indicates that data item B is to be located via the pointer variable A. Such expressions are called locating expressions. The above expression might be used in an assignment statement as follows:
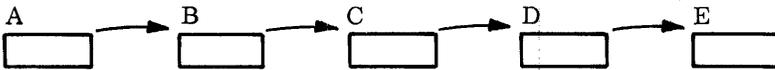
```
TOTAL = A -> B;
```

The statement would read "assign the value of data item B, as located by the pointer variable A, to the variable TOTAL."

The relationship between A and B might be shown graphically as follows:

A   B

Suppose that B is also a pointer variable, however, and that several other pointer variables are also linked in a chain, ending eventually at data item E:

A   B   C   D   E

The locating expression would then be:

```
A -> B -> C -> D -> E
```

Now assume that each pointer variable except the first is actually an entry in a table, and that the tables are thereby arranged in a chained list of non-contiguous tables, all of which have the same format:

A

ADDNEXT		ADDNEXT		ADDNEXT		ADDNEXT
INFO		INFO		INFO		INFO
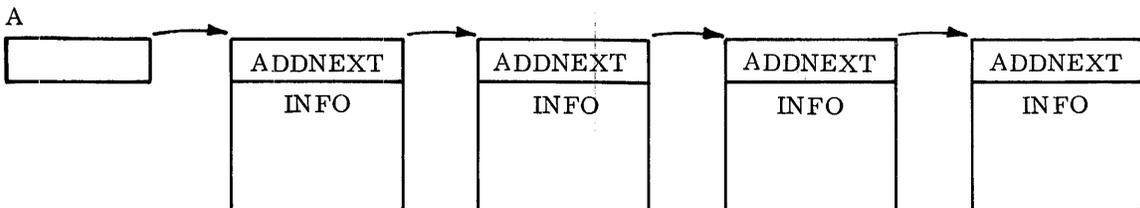
Two DECLARE statements would be needed to describe the above arrangement to the BSL compiler:

```
DECLARE A POINTER;
DECLARE 1 TABLE BASED (A),
    2 ADDNEXT POINTER,
    2 INFO CHARACTER (12);
```

The first DECLARE statement identifies A as being a pointer variable.

The second DECLARE statement describes a single item, TABLE. Because all the tables have the same format, all symbolic names will apply equally to any table in the chain. The BASED attribute indicates to the compiler that the pointer variable A contains the address of the first table of the chain.

Following the above declarations, the first field, ADDNEXT, of the last table in the chain could be referred to with the following expression:

```
A -> ADDNEXT -> ADDNEXT -> ADDNEXT -> ADDNEXT
```

Similarly, the second field, INFO, of the third table in the chain could be referred to as follows:

```
A -> ADDNEXT -> ADDNEXT -> INFO
```

The above expression is valid because the BSL compiler, using information derived from the DECLARE statement, will automatically compute the displacement of INFO from the beginning of the appropriate table.

For a practical application, first assume that two tables are chained as follows:
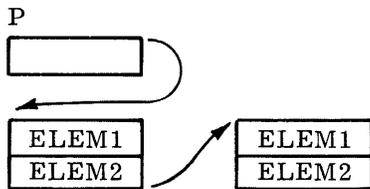
Pointer variable P contains the address of the first table, which in turn contains the address of the second table. Assume further that another table exists, which is similar in format, but which is pointed to by Q:
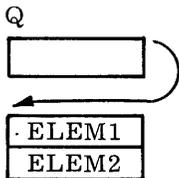
Now, suppose that the single table is to be added to the chain between the two tables currently in the chain. First, the pointer variables P and Q must be declared:
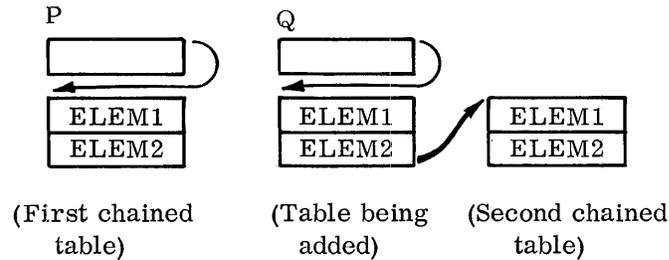
```
DECLARE (P, Q) POINTER;
```

Next, the format of the tables must be declared:

```
DECLARE 1 TABLE BASED(P),
     2 ELEM1 CHARACTER (4),
     2 ELEM2 POINTER;
```

Now, with the following assignment statement, the address of the second of the chained tables can be placed into the table being added to the chain:

```
Q -> ELEM2 = P -> ELEM2;
```

The statement reads "assign the value of ELEM2, as located by the pointer variable P, to ELEM2, as located by the pointer variable Q." The results of the operation are shown below:

P   Q

| ELEM1 | | ELEM1 | | ELEM1 |
| ELEM2 | | ELEM2 | | ELEM2 |

(First chained   (Table being   (Second chained
    table)   added)   table)

As can be seen from the assignment statement, the needed address was obtained from the first chained table, which still points to the same place, but can now be made to point to the table being added to complete the chain:

```
P -> ELEM2 = Q;
```

The statement reads "assign the value of Q to ELEM2, as located by the pointer variable P." The results of the operation are shown below:

P   Q

| ELEM1 | | ELEM1 | | ELEM1 |
| ELEM2 | | ELEM2 | | ELEM2 |
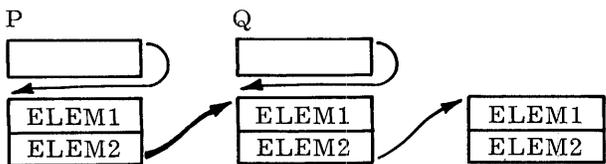
When an item is declared to have the BASED attribute, it is not necessary to use a locating expression in referring to the item. For example, if the declarations

```
DECLARE A POINTER;
DECLARE B BASED (A);
```

are written, the expression A - > B need not be written when a reference to B is to be made. The name B may be used alone, as in the following statement:

```
ITEMA = B;
```

Thus, if several levels of indirect addressing are in use, it is possible to refer directly to an item as follows:
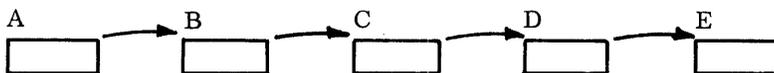

```
DECLARE A POINTER, B POINTER BASED (A), C POINTER BASED (B),
        D POINTER BASED (C), E BASED (D);
ITEMA = E;
```

Locating expressions need not reflect relationships that have been declared. For example, the declaration

```
DECLARE B BASED (A);
```

indicates that the address of item B is contained in item A. If it happens that another item, say C, also contains the address of B, the following expression might then be written in referring to B:

```
C -> B
```

CHECKPOINT 3: PROBLEM

Assume that the three chained tables shown below exist, and that the address of the first table is contained in storage location A.

A

ADDNEXT	ADDNEXT	ADDNEXT (Zeros)
ADDLAST (Zeros)	ADDLAST	ADDLAST
DATA	DATA	DATA
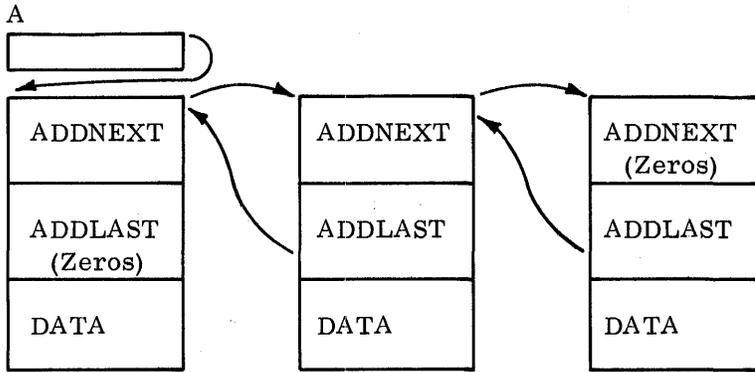
Assume also that another table having the same format as the tables in the chain exists, but that the address of this table is contained in storage location B:

B

DATA
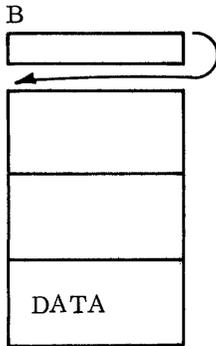
Write the statements necessary to add the new table to the chain of tables between the second and third tables in the chain.

A solution to this problem appears on the following page.

```
/* SOLUTION TO CHECKPOINT 3 PROBLEM */

/* DESCRIBE FORMAT OF TABLES */
        DECLARE (A, B) POINTER;
        DECLARE 1 TABLE BASED (A),
                2 ADDNEXT POINTER,
                2 ADDLAST POINTER,
                2 DATA CHARACTER (4);
/* ADD TABLE TO CHAIN */
        B -> ADDLAST = A -> ADDNEXT -> ADDNEXT -> ADDLAST;
        A -> ADDNEXT -> ADDNEXT -> ADDLAST = B;
        B -> ADDNEXT = A -> ADDNEXT -> ADDNEXT;
        A -> ADDNEXT -> ADDNEXT = B;
```

The numbered lines in the diagram shown below indicate the order in which the operations are performed; the broken lines indicate the original arrangement of the tables.

Table Being
Added
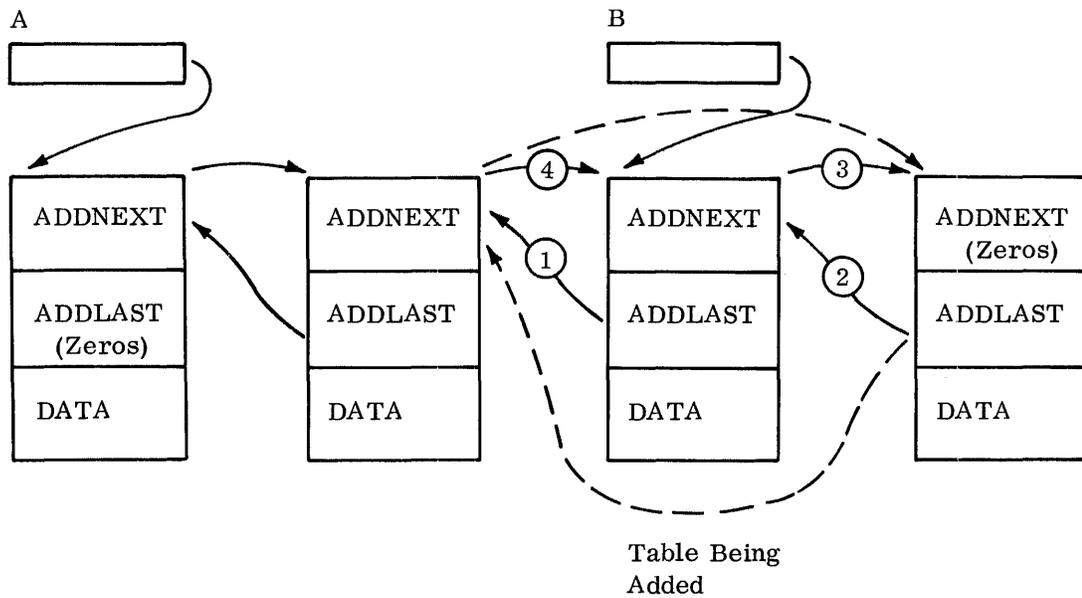
Often, it is desirable to divide a program into a number of parts. It might be that several programmers are working with the same program but, for practical reasons, each must code and debug his own assignment separately. Or, to make a program more manageable and easier to debug, a single programmer may elect to divide his assignment into two or more parts, and then to test each part separately. With BSL, programs may be divided into any number of parts, called procedures, each of which may be compiled, assembled, and executed separately for test purposes, and then combined and executed as a single unit.

BSL programs do not have to be divided into two or more procedures, as an entire program can be made to consist of only one procedure. The choice of how many procedures to have and where a division is to be made is entirely up to the programmer.

Two BSL statements, the PROCEDURE statement and the END statement, are used to mark the beginning and the end of each procedure. The PROCEDURE statement must be preceded by a symbolic name, and the entire procedure may be referred to by that procedure name:

```
CHECKTCB: PROCEDURE;
```

The above statement marks the beginning of procedure CHECKTCB. An END statement must be the last statement of each procedure. It can consist of only the keyword END, but may also contain the procedure name:

```
END CHECKTCB;
```

Single-part programs require only one PROCEDURE statement and one END statement:

```
CHECKTCB: PROCEDURE;
          statement 1;
          statement 2;
          statement 3;
          END CHECKTCB;
```

The above procedure, CHECKTCB, contains three statements (besides the PROCEDURE and END statements). A procedure may contain any number of statements, however.

When the decision to divide a program into two or more procedures has been made, several things must be considered. One consideration is the flow of control, or the order in which the procedures will be executed and the points at which each will be entered. Another consideration involves data; some data can be made accessible to all procedures, while other data can be made to be more limited in scope. Storage allocation must also

be considered. When data is being shared by two or more procedures, the space it is to occupy must be set aside in one of those procedures.

The above considerations could not only have a direct bearing upon where a division is made, but could also influence the ways in which the individual procedures are written.

FLOW OF CONTROL CONSIDERATIONS

Control is not passed automatically from one procedure to the next. A transfer of control must be indicated explicitly with a CALL statement containing the name of the entry point of the procedure to be entered, or invoked:

```
CALL CHECKTCB;
```

The above statement will cause entry to procedure CHECKTCB. CALL statements may be placed at any point from which a transfer to another procedure is desired:

```
        MAIN:   PROCEDURE;
                statement 1;
                statement 2;
                CALL CHECKTCB;
                statement 4;
                END MAIN;

CHECKTCB:   PROCEDURE;
                statement a;
                statement b;
                END CHECKTCB;
```

In the above example, execution begins with statement 1 of MAIN. Then statement 2 is executed. The CALL CHECKTCB statement invokes the procedure named CHECKTCB, and control is transferred to CHECKTCB, where statements a and b are executed. When the END CHECKTCB statement is reached, control is returned to procedure MAIN, where the statement immediately following the CALL CHECKTCB statement is executed.

When CHECKTCB is called, it is the invoked procedure and MAIN is the invoking procedure; the CALL CHECKTCB statement is the point of invocation.

As procedure CHECKTCB now stands, it can be invoked at only one point, CHECKTCB. Therefore, it has only one entry point. Additional entry points could be designated with ENTRY statements, which consist of the keyword ENTRY preceded by a symbolic statement name which is to serve as the entry point name:

```
CHANGE: ENTRY;
```

Any number of ENTRY statements are allowed in a procedure, and they may be placed wherever entry points are desired. The following example illustrates the use of an ENTRY statement:

```
MAIN:     PROCEDURE;
          statement 1;
          statement 2;
          CALL CHANGE;
          statement 4;
          END MAIN;

CHECKTCB: PROCEDURE;
          statement a;
          CHANGE:  ENTRY;
          statement c;
          END CHECKTCB;
```

The CALL CHANGE statement invokes procedure CHECKTCB at point CHANGE, where statement c is executed. Execution of the END CHECKTCB statement then results in control being returned to MAIN at the statement following the point of invocation, or statement 4.

Should it be desirable to cause control to be returned to an invoking procedure before the END statement of the invoked procedure is reached, a RETURN statement can be used. The RETURN statement consists only of the keyword RETURN, followed by a semicolon delimiter:

```
RETURN;
```

The RETURN statement, like an END statement, will cause control to be returned to the invoking procedure at the statement following the point of invocation. To illustrate, a RETURN statement might be used as follows:

```
MAIN:     PROCEDURE;
          statement 1;
          statement 2;
          CALL CHECKTCB;
          statement 4;
          END MAIN;

CHECKTCB: PROCEDURE;
          statement a;
          statement b;
          RETURN;
          statement d;
          END CHECKTCB;
```

In the above, the CALL CHECKTCB statement invokes procedure CHECKTCB, where statements a and b are executed. When the RETURN statement is executed, control will be returned to procedure MAIN at the statement following the point of invocation, or statement 4.

Control can be returned to any named statement of an invoking procedure if a RETURN TO statement is used:

```
RETURN TO VALIDTCB;
```

With the above, control will be returned to the invoking procedure at the statement named VALIDTCB. The name of the statement to which control is to be returned must be declared to have the attributes LABEL, LOCAL, and EXTERNAL in the invoking procedure. The name must also be declared to have LABEL and EXTERNAL attributes in the invoked procedure. The LOCAL and EXTERNAL attributes are discussed in the section Data Considerations; the need for an explicit LABEL attribute will be explained after the following example, in which use of a RETURN TO statement is illustrated:

```
      MAIN:  PROCEDURE;
             DECLARE VALIDTCB LABEL LOCAL EXTERNAL;
             statement 2;
             statement 3;
             CALL CHECKTCB;
             statement 5;
             VALIDTCB:  statement 6;
             statement 7;
             END MAIN;

  CHECKTCB:  PROCEDURE;
             DECLARE VALIDTCB LABEL EXTERNAL;
             statement b;
             statement c;
             RETURN TO VALIDTCB;
             statement e;
             END CHECKTCB;
```

With the above, procedure CHECKTCB will be invoked when the CALL CHECKTCB statement in procedure MAIN is executed. Then, when the RETURN TO VALIDTCB statement is executed, control will return to the statement named VALIDTCB of procedure MAIN.

Now observe that, had the LABEL attribute been omitted from either declaration in the above example, the default attributes for the item VALIDTCB would have been FIXED (31). Thus, VALIDTCB would have been taken to be a signed, arithmetic item--the LABEL attribute ensures that VALIDTCB would be treated as a label.

DATA CONSIDERATIONS

If the name and attributes of a data item are declared in one procedure, it does not always follow that the name can be used in other procedures. Furthermore, the same name can be declared in two or more procedures, but may be taken to represent different items of data in each. In a single-procedure program, names are known throughout the program; their scope is the entire program. In multi-procedure programs, however, the scope of a name may be more limited. The scope of a name is the range of the program throughout which the name is known.

Three methods exist by which the scope of a name may be extended. Procedures can be arranged one inside the other, or nested, in such a way that names known to the encompassing procedure are known also to the nested procedure. In addition, certain scope attributes can be specified in the DECLARE statement to identify items whose scope is to be extended to include procedures other than the declaring procedure. With the third method, the scope of names can be extended by passing arguments consisting of the names of data items from one procedure to other procedures in which the same items will be used.

NESTING PROCEDURES

A nested procedure is one that is wholly contained within another procedure. In the following example, two identical procedures are illustrated. In the first program they are arranged consecutively, while in the second program, one is nested within the other:

```
A:  PROCEDURE;                       A:  PROCEDURE;
      statement 1;                         statement 1;
      statement 2;                         statement 2;
      CALL B;                              CALL B;
      END A;                          B:  PROCEDURE;
                                           statement a;
B:  PROCEDURE;                             statement b;
      statement a;                         END B;
      statement b;                         END A;
      END B;
```

Control flow will be the same for both programs. From external appearances, the only change appears to be one of physical organization. However, another, more important, difference now exists.

Any data names declared in procedure A of the first example are known only to that procedure. In fact, names used in A could be repeated in B, but would be taken to represent different data items, even though their attributes might be declared to be identical.

On the other hand, in the second of the above examples, any names declared in procedure A are known also to procedure B. Consequently, any name used in A can also be used in B.

The relationship between the nested procedures A and B might be shown as follows:

A _____
┌─────────────────
│ B _____
│ ┌────────────
│ │ _____
│ └────────
│
└──────────────
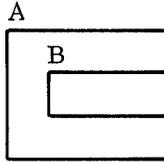
Several levels of nesting could be used, and two or more procedures can appear at the same level. To illustrate, two additional procedures, C and D, might be added in either of the following ways:

A _____   A _____
┌─────────────────   ┌─────────────────
│ B _____   │ B _____
│ ┌───────────────   │ ┌───────────────
│ │ C _____   │ │ C _____
│ │ ┌─────────────   │ │ ┌─────────────
│ │ │ D _____   │ │ │ _____
│ │ │ ┌───────────   │ │ └──────────
│ │ │ │ _____   │ │ D _____
│ │ │ └─────────   │ │ ┌─────────────
│ │ └───────────   │ │ │ _____
│ └─────────────   │ │ └──────────
└───────────────   │ └──────────────
                            └────────────────

Data names used in a procedure are known only to the procedure in which they are declared and to any lower-level nested procedures. In both of the above examples, a name declared in B would be known to C and D, but not to A. In the first example, a name declared in C would be known to D, but not to A or B; in the second example, a name declared in C would be known only to C.

Nested procedures may be invoked only from the immediately encompassing procedure or any procedure at the same level. In the first of the above examples, a CALL C statement would be valid only in procedure B, while in the second example a CALL C statement would be valid in B or D.

A single nested procedure must always be positioned so that its END statement immediately precedes the END statement of the containing procedure. Multiple procedures at the same level must be adjacent to each other, the END statement of the last placed just before the END statement of the containing procedure. For example, procedures A, B, C, and D shown above would be arranged as follows:

```
A: PROCEDURE;                          A: PROCEDURE;
        .                                      .
        .                                      .
        .                                      .
    B: PROCEDURE;                          B: PROCEDURE;
            .                                      .
            .                                      .
            .                                      .
        C: PROCEDURE;                          C: PROCEDURE;
                .                                      .
                .                                      .
                .                                      .
                                               END C;
            D: PROCEDURE;                      D: PROCEDURE;
                    .                                  .
                    .                                  .
                    .                                  .
                END D;                             END D;
            END C;                             END B;
        END B;                             END A;
    END A;
```

When procedures are nested, the entire nest may be compiled as one unit. In all other cases, procedures must be compiled separately.

USING SCOPE ATTRIBUTES

To review, the scope of a data name is the range of a program throughout which the name is known. It has been shown how the scope of a name can be changed by placing procedures into a nested arrangement. The scope of a name may also be changed, however, by the use of scope attributes in the DECLARE statement.

There are two scope attributes: INTERNAL and EXTERNAL. The INTERNAL attribute serves to restrict the scope of a name to the procedure containing the declaration; the same name declared to be INTERNAL in two or more procedures would be taken to mean different data items in each of the procedures.

The EXTERNAL attribute serves to extend the scope of a name from the procedure containing the declaration to any other procedure containing the same declaration. The same name declared to be EXTERNAL in two or more procedures would be taken to mean the same data item.

In the first of the following examples, the name TOTAL represents two different data items, while in the second example, it represents the same data item:

```
A:  PROCEDURE;                          A:  PROCEDURE;
        DECLARE TOTAL                           DECLARE TOTAL
            FIXED INTERNAL;                         FIXED EXTERNAL;
            .                                       .
            .                                       .
            .                                       .
        END A;                                  END A;

B:  PROCEDURE;                          B:  PROCEDURE;
        DECLARE TOTAL                           DECLARE TOTAL
            FIXED INTERNAL;                         FIXED EXTERNAL;
            .                                       .
            .                                       .
            .                                       .
        END B;                                  END B;
```

PASSING ARGUMENTS

A data item known to an invoking procedure can be made known to the invoked procedure
if its name is passed as an argument of invocation. This is done by placing the data
name into a CALL statement as follows:

```
CALL B (ONE);
```

The name may then be placed similarly into either a PROCEDURE statement named B
or an ENTRY statement named B as follows:

```
B:  PROCEDURE (ONE);
```
```
B:  ENTRY (ONE);
```

Data names so specified in CALL statements are called arguments; data names so spec-
ified in PROCEDURE or ENTRY statements are called parameters. More than one name
may be passed, but the names must be separated by commas and parentheses must en-
close both the argument list and the parameter list:

```
A:  PROCEDURE;
        CALL B (ONE, TWO, THREE);
            .
            .
            .
        END A;

B:  PROCEDURE (ONE, TWO, THREE);
            .
            .
            .
        END B;
```

IBM CONFIDENTIAL

In the preceding example, B is invoked by the CALL statement in A, and the names of data items ONE, TWO, and THREE are passed from A to B; that is, their scope is extended from A to include B.

Names in an argument list are associated with names in a parameter list only by the order in which they appear; that is, the first argument is associated with the first parameter, the second argument with the second parameter, etc. No association by name is made; in fact, for nested procedures, the name of a parameter must not be the same as the name of the corresponding argument. The preceding example could just as well have been expressed as follows:

```
A:  PROCEDURE;
    CALL B (ONE, TWO, THREE);
        .
        .
        .
    END A;

B:  PROCEDURE (FOUR, FIVE, SIX);
        .
        .
        .
    END B;
```

With the above, the item named ONE would be referred to in procedure A by the name ONE and in procedure B by the name FOUR; each name represents the same data item.

Consider the following example, in which a subroutine, C, is invoked by procedures A and B:

```
A:  PROCEDURE;                          B:  PROCEDURE;
    DECLARE INDICATR BIT (24);              DECLARE SWITCH BIT (24);
        .                                       .
        .                                       .
        .                                       .
    CALL C (INDICATR);                      CALL C (SWITCH);
        .                                       .
        .                                       .
        .                                       .
    END A;                                  END B;
              C:  PROCEDURE (INPUT);
                  DECLARE INPUT BIT (24);
                      .
                      .
                      .
                  INPUT = 5;
                      .
                      .
                      .
                  END C;
```

In the example, C is a procedure that is used as a subroutine by both A and B. A and B each share a data item with C, but not with each other. Although the names of the items differ, the attributes of the items passed must agree, and they must also be declared in all procedures involved. With the CALL statements, both the scope of INDICATR and the scope of SWITCH are extended to include procedure C. When A invokes C, INDICATR will be assigned a value of 5; when B invokes C, SWITCH will be assigned a value of 5.

Arguments may consist of constants, variables, or complex expressions, and the variables may be structure or array names.

STORAGE ALLOCATION CONSIDERATIONS

Data names do not actually occupy storage space during execution, as they are converted to addresses prior to execution. The data items at the locations, however, do occupy space and this space can be an important consideration. For example, in a multi-procedure program in which data is being shared, the procedure in which the shared items are to be assigned space should be made known to the compiler--otherwise no space will be assigned. Two storage class attributes, LOCAL and NONLOCAL, provide a means of indicating the procedure in which a particular item is to be assigned space. Also, if certain items are referred to in only one procedure, space could be saved if the items are not assigned space until the procedure is invoked, and if the space is freed upon exit from the procedure. The two storage class attributes STATIC and AUTOMATIC provide a means for indicating permanent and temporary space assignment.

The LOCAL attribute indicates that space is to be assigned to an item, while the NONLOCAL attribute indicates that space is not to be assigned to an item:

```
A:  PROCEDURE;
    DECLARE TABLE (12) CHARACTER (10)
                  LOCAL EXTERNAL;

          .
          .
          .
    END A;

B:  PROCEDURE;
    DECLARE TABLE (12) CHARACTER (10)
                  NONLOCAL EXTERNAL;

          .
          .
          .
    END B;
```

In the example shown above, the array TABLE is to be shared by procedures A and B. TABLE will be assigned space once--in the storage area assigned to procedure A.

The default storage class attribute may be LOCAL or NONLOCAL, as it depends upon the scope attribute that is specified. If the scope of an item is declared to be IN-TERNAL, the default storage class attribute will be LOCAL. However, if the scope of

an item is declared to be EXTERNAL, the default storage class attribute will be NON-
LOCAL:

Scope Attribute	Default Storage Class Attribute
INTERNAL EXTERNAL	LOCAL NONLOCAL

If neither a scope attribute nor a storage class attribute is specified, the default attri-
butes will be INTERNAL and LOCAL.

When the STATIC attribute is specified for a data item, space is assigned perma-
nently to that item during compilation:

```
A:  PROCEDURE;
      DECLARE TABLE (12) CHARACTER (10)
                STATIC;
           .
           .
           .
      END A;
```

With the above, the space assigned to the array TABLE will be obtained during compi-
lation; it will not be made available for reassignment to any of the items declared in
other procedures of the program. The STATIC attribute is a default attribute. There-
fore, it need not be explicitly specified as it was in the preceding example.

If the AUTOMATIC attribute is specified for a data item, space will not be obtained
for the item until the procedure containing the declaration is invoked. Then, upon exit
from the procedure, the space will be freed for possible reassignment. Whenever the
AUTOMATIC attribute is specified, neither the LOCAL nor the NONLOCAL attribute
may be specified. Also, the PROCEDURE statement for the procedure in which the dec-
laration is made must contain the keywords OPTIONS (REENTRANT). The following ex-
ample illustrates how the AUTOMATIC storage class attribute might be used:

```
A:  PROCEDURE OPTIONS (REENTRANT);
      DECLARE TABLE (12) CHARACTER (10)
                AUTOMATIC;
           .
           .
           .
      END A;
```

With the above, space will not be obtained for the array TABLE until procedure A is in-
voked; the space will be freed upon exit from A.

CHECKPOINT 4: PROBLEM

In the checkpoint 3 problem, it was said to assume that three chained tables existed, and that the address of the first of the chained tables was contained in storage location A. The format and arrangement of the tables was shown to be:

A

ADDNEXT	ADDNEXT	ADDNEXT (zeros)
ADDLAST (Zeros)	ADDLAST	ADDLAST
DATA	DATA	DATA
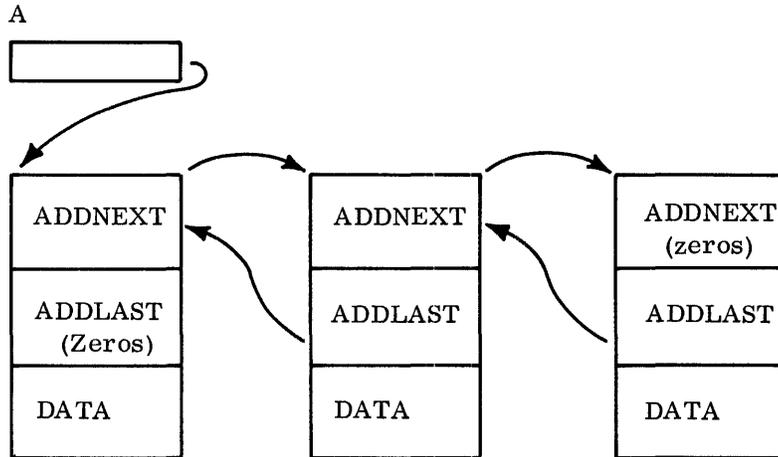
It was also said to assume that another table of the same format existed, but that the address of this table was contained in storage location B:

B

DATA
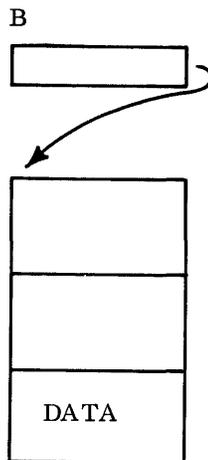
Write a procedure called BUILD, and in it construct both the chain of tables and the single table to be added to the chain. Pass the address of the first table in the chain and that of the table to be added to the chain to another procedure, UPDATE.

Add PROCEDURE and END statements to the statements consituting the solution to the checkpoint 3 problem. Call the procedure UPDATE, and have it receive, as parameters, the addresses of both the first table in the chain and the table to be added.

A solution to this problem appears on the following page.

```
/* SOLUTION TO CHECKPOINT 4 PROBLEM */

BUILD:  PROCEDURE;
        DECLARE (A, B) POINTER;
        DECLARE 1 TABLE (4),
                2 ADDNEXT POINTER,
                2 ADDLAST POINTER,
                2 DATA CHARACTER (4);
/* CHAIN THREE TABLES */
        A = ADDR (TABLE (1));
        ADDNEXT (1) = ADDR (ADDNEXT (2));
        ADDNEXT (2) = ADDR (ADDNEXT (3));
        ADDNEXT (3) = 0;
        ADDLAST (1) = 0;
        ADDLAST (2) = ADDR (ADDNEXT (1));
        ADDLAST (3) = ADDR (ADDNEXT (2));
/* ADDRESS OF FOURTH TABLE TO B */
        B = ADDR (TABLE (4));
/* PASS A AND B TO PROCEDURE UPDATE */
        CALL UPDATE (A, B);
        END BUILD;

UPDATE:  PROCEDURE (A, B);
        statements for problem 3
            .
            .
            .
        END UPDATE;
```

CHAPTER 5: CONTROLLING PROGRAM FLOW

Normally, statements are executed one by one in the order by which they appear in a program. Thus far, it has been shown how that order can be changed with the CALL, END, RETURN, and RETURN TO statements. The order of execution can also be changed, however, with the IF, GO TO, and DO statements. With these statements, a change can be conditional or unconditional, or statements can be grouped and treated as a group, to be executed together or skipped entirely.

CONDITIONAL CHANGES TO PROGRAM FLOW

The IF statement is used when the order of execution is to depend upon a relationship that exists between two or more data items. The relationship is expressed, and, depending upon whether that relationship is true or false, a choice of designated paths to be taken is made.

Two keywords constitute the basic IF statement, IF and THEN. In skeletal form the basic IF statement can be shown as follows:

```
IF _____ THEN _____ ;
```

The relationship between two data items is expressed after the IF keyword, and the action to be taken if the relationship is true is expressed after the THEN keyword. If the relationship is false, the action is not taken. The example shown below illustrates how a basic IF statement can be used.

```
IF A > B THEN A=C;
CALL SUBRTN;
```

The IF statement reads "If the value of A is greater than that of B, then assign the value of C to A." The CALL SUBRTN statement is used for illustrative purposes; it represents any statement that may follow the IF statement. It will be executed regardless of whether or not the value of A is greater than that of B.

The expression A > B is called a relational expression. Relational expressions always show a comparison that is to be made, and may only be used in IF statements. The "greater than" sign is a comparison operator, one of a group of operators that designate the way in which the specified items are to be compared (see Figure 4). Items being compared may be constants or variables.

Either of two paths may be taken in the preceding example. If the relationship is true, that is, if the value of A is greater than that of B, then the assignment statement A=C will be executed, and control will pass to the statement following the IF statement, CALL SUBRTN:

```
      YES _____
IF A > B THEN A=C;
CALL SUBRTN;
```
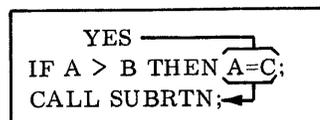
```
  >        Greater than

  <        Less than

  =        Equal to

 ⌐>        Not greater than

 ⌐<        Not less than

 ⌐=        Not equal to

 >=        Greater than or equal to

 <=        Less than or equal to

All comparison operators have a
priority of 4.
```

Figure 4. Comparison Operators

However, if the relationship is false, the assignment statement A=C will be skipped, and control will pass directly to the CALL SUBRTN statement:

```
     NO ─────────────┐
IF A > B THEN A=C;    │
CALL SUBRTN;◄─────────┘
```
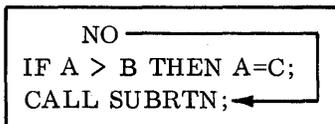
The statement following the THEN keyword can be any statement except the DE-CLARE, END, ENTRY, and PROCEDURE statements, all of which have been discussed, and the RESTRICT and RELEASE statements, discussed in the publication BSL: Basic Systems Language Description.

As the example now stands, there are two paths for control flow to follow. A third path may be added with an ELSE clause:

```
     NO ──────────────────┐
IF A > B THEN A=C; ELSE A=D;
CALL SUBRTN; ◄─────────────┘
```
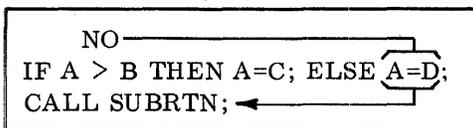
Now, if the relationship between A and B is false, the statement A=D will be executed before control passes to the CALL SUBRTN statement.

An ELSE clause always designates an action to be taken if a relationship expressed in an IF statement is false. ELSE clauses may only be used with IF statements; they are optional but, when used, they must follow the semicolon that delimits the basic IF statement, and they must themselves be followed by a semicolon delimiter.

In all examples discussed thus far, the actions to be taken have depended upon the result of one relationship, expressed in a single relational expression. It has been mentioned that the statement following the THEN keyword can be any statement except

IBM CONFIDENTIAL

57

DECLARE, END, ENTRY, PROCEDURE, RESTRICT, or RELEASE. Therefore, the statement following the THEN keyword of one IF statement can be another IF statement. Furthermore, the statement following the THEN keyword of the second IF statement can also be an IF statement, etc. This allows not one, but any number of relationships to be established, with separate true-false paths for each. The following is an example of such a nested IF arrangement:
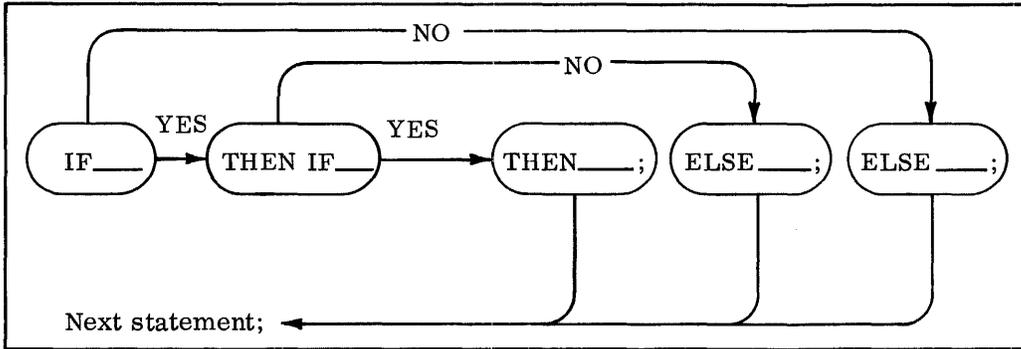
One nested IF statement is shown. Notice the association of IF statements and ELSE clauses (the "no" paths). If the first relationship is false, control passes to the last ELSE clause; if the second relationship is false, control passes to the second-to-last ELSE clause. If there were a third nested IF statement and the relationship was false, control would pass to the third-to-last ELSE clause.

The preceding example is shown in Figure 5 as it might appear in a program. The free-form format facilities of BSL are used to advantage, as the ELSE clauses are made to appear directly below their associated IF statements.

It was mentioned earlier that ELSE clauses are optional. When there are no ELSE clauses in a nested IF arrangement, all "no" paths will lead to the next statement. If an equal number of IF statements and ELSE clauses exists, the IF-ELSE relationship is that which has been shown above (first to last, etc.). However, when there is an imbalance, as when there are fewer ELSE clauses then IF statements, each ELSE clause is automatically paired with the nearest IF statement. To illustrate, if one ELSE clause were omitted from the nested IF arrangement in the example shown above, the relationship would change to the following:
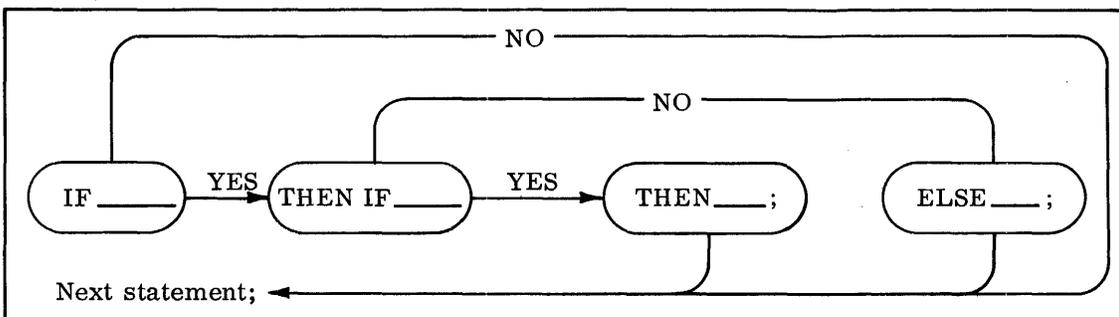
Now if the first relationship is false, control will pass to the next statement--not to the last (in this case, the only) ELSE clause.
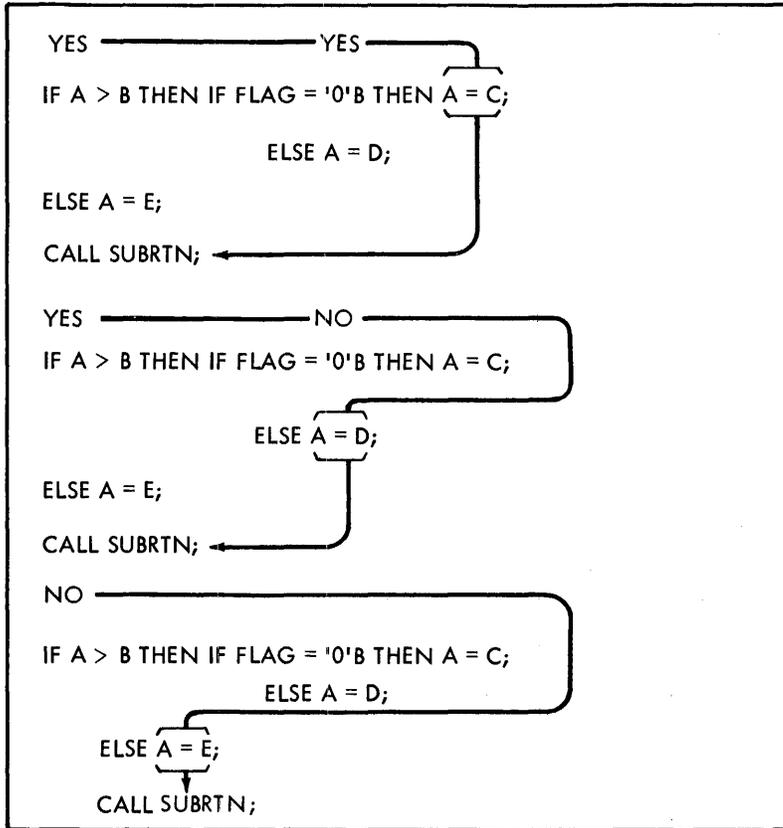
Figure 5. Nested IF Statement- ELSE Clause Relationships

When a "no" action is not desired but a specified IF-ELSE relationship must be preserved, a null ELSE clause may be used. It consists simply of the keyword ELSE, followed by a semicolon delimiter:

```
ELSE;
```

Null ELSE clauses cause no change in control flow. Thus, in Figure 5, a null ELSE clause could be substituted for any of the ELSE clauses shown, and no other change to the figure would be necessary.

The logical operators & (and), | (or), and && (exclusive or) can be used in relational expressions. For example, the statement

```
IF A > =(B&C) THEN A=C;
```

reads "If the value of A is greater than or equal to that which will result when B is "anded" to C, then assign C to A." The parentheses are needed because the priority of the > = operator is greater than that of the & operator; without the parentheses shown above, the statement would be meaningless, as it would be interpreted to be:

```
IF (A > =B)&C THEN A=C;
```

The two operators & (and) and | (or) can also be used when an action is to depend upon a relationshipship between two or more relational expressions. For example, the statement

```
IF A>B|C<D THEN A=C;
```

reads "If the value of A is greater than that of B, or if the value of C is less than that of D, then assign C to A." When used in this way, the operators are called connectives, to distinguish them from the logical operators which consist of the same symbols.

UNCONDITIONAL CHANGES TO PROGRAM FLOW

The GO TO statement, like the IF statement, can be used to alter the normal order of execution of a program. Unlike the IF statement, however, the GO TO statement results in an unconditional, rather than a conditional change in the order of execution.

The GO TO statement consists of the keywords GO TO, followed by the name of the statement to which control is to be passed:

```
GO TO ALPHA;
   .
   .
   .
ALPHA:  A=B;
```

In the above example, control will be transferred unconditionally from the GO TO statement to the statement named ALPHA; all intervening statements will be skipped.

Although a GO TO statement by itself results in an unconditional transfer of control, it can be used in an IF statement, where it would be executed conditionally. Thus, the net result would be a conditional transfer of control:

```
IF A>B THEN GO TO UPDATE;
   .
   .
   .
UPDATE:  A=C;
```

In the above example, control would be transferred from the IF statement to the statement named UPDATE only if the value of A is greater than that of B. GO TO statements may follow both THEN and ELSE keywords.

GROUPING STATEMENTS

The DO statement provides a means of grouping a number of statements in such a way that they will be treated as a single unit. The unit, called a DO group may then be used in two ways. The group may be placed after the THEN or ELSE keywords of an IF statement, in which case the group will be executed or skipped entirely, depending upon the results of a comparison. The group of statements may also be executed repeatedly a specified number of times before control passes to the statement following the group.

A DO statement must always be the first statement of a DO group, and an END statement must always be the last statement of the group. In its simplest form, the DO statement consists of only the keyword DO and a semicolon delimiter, but it may also be given a statement name:

```
        DO;
```

```
SUBRTN:  DO;
```

The END statement can consist of only the keyword END, but, if the DO statement is given a name, the END statement may also contain that name:

```
END;
```

```
END SUBRTN;
```

Any group of statements bounded by the above DO and END statements would be treated as a single unit. Thus, the group could appear in an IF statement as follows:

```
IF A>B THEN DO;
        statement 1;
        statement 2;
        statement 3;
        END;
ELSE statement 4;
statement 5;
```

Statements 1, 2, and 3 will be executed only when the value of A is greater than that of B. The END statement signifies the end of the DO group and, in this case, the end of the THEN clause. Therefore, when the END statement is reached, control will pass to statement 5. When the value of A is less than that of B, the DO group will be bypassed entirely and the ELSE clause will be executed.

A DO group can be made to be executed repeatedly a designated number of times if certain additional information is placed into the DO statement. In principle, the compiler establishes a counter and increments the count each time a pass is made through the DO group. The programmer must specify: a symbolic name to represent the counter; the initial value to be placed into the counter; and a value which, when exceeded, will result in exit from the group.

The skeletal form of a DO statement that can be used to cause repetitive execution of a DO group is:

```
DO _____ TO _____ ;
```

Both the variable name that is to represent the counter and the initial value that the counter is to have are expressed after the DO keyword in the form of an assignment statement. The value to be exceeded before exit is made from the DO group is specified after the TO keyword. For example, a DO group that is to be executed 10 times before control is to pass to the statement following the group might be expressed as follows:

```
DO A=1 TO 10;
        statement 1;
        statement 2;
        statement 3;
        END;
```

In the above, the variable named A represents the counter, which is given an initial value of one. Any symbolic name may be used to represent the counter. The counter will be incremented automatically by one each time the DO group is executed. When its value reaches 11, or, in other words, after the group has been executed 10 times, control will pass to the statement following the END statement.

Constants, variables, or arithmetic expressions may be used to designate both the initial value of the counter and the value to be exceeded when exit from the group is desired. Thus, the DO statement in the preceding example could just as well have been written as follows:

```
DO A=B+C TO NO;
```

In the above, the number of iterations would depend upon the values of the variables B, C, and NO.

In the examples shown thus far, the counter was incremented by one each time the DO group was executed. Other values by which the counter is to be incremented can also be specified, however, by adding the keyword BY to the DO statement, and following the BY keyword with the desired increment, as shown in the following example:

```
DO A=0 TO 10 BY 2;
      statement 1;
      statement 2;
      statement 3;
      END;
```

The above DO group would be executed six times. The increment, in this case two, could also be expressed as a variable or an arithmetic expression. If an increment is not specified, an increment of one will be used.

A negative increment may be specified in the DO statement, in which case the initial value given the counter will be decremented. Exit from the group will occur when the value of the counter becomes less than that specified after the TO keyword:

```
DO A=10 TO 1 BY -1;
      statement 1;
      statement 2;
      statement 3;
      END;
```

The DO group shown above will be executed 10 times; the counter will have a value of 0 upon exit from the group.

DO groups, like procedures and IF statements, may be nested; that is, a statement of a DO group may be another DO statement, as in the following example:

```
DO A=1 TO 10;
      statement 1;
      statement 2;
      DO B=1 TO 5;
           statement a;
           statement b;
           END;
      END;
```

In the above example, the second DO group will be executed five times for each execution of the first DO group, or a total of 50 times. Care should be taken with nested DO groups to ensure that each DO group has an associated END statement.

The automatically decrementing or incrementing value of a counter may often be used to advantage, for the symbolic name given the counter may be used to denote a variable in other statements of the program. For example, assume that an array of 12 items exists, and that the position of the first item containing zeros is to be determined. The following statements might be written:

```
        NO = 0;
        DO A = 1 TO 12;
                IF ARRAY (A) = 0 THEN DO;
                    NO = A;
                    GO TO OUT;
                    END;
        END;
OUT:  IF NO = 0 THEN...;
```

In the above, the subscript of ARRAY will reflect the value of the counter. The first time the group is executed, A will be 1 and the first item in the array will be checked for zeros; the second time the group is executed, A will be 2 and the second item will be checked; etc. When an item containing zeros is encountered, a value representing the position of the item in the array will be assigned to NO, and exit from the group will occur. Should none of the items in the array contain zeros, the value of NO will remain 0. Thus, upon exit from the group, the variable NO will have a value of 0 if none of the items contained zeros, or a value that reflects the position of the first element in the array which contains zeros.

Because the counter is reset automatically each time that a DO statement is reached, a transfer out of any DO group, as with a GO TO statement, will not affect the operation the next time the group is executed.

CHECKPOINT 5: PROBLEM

Write two procedures. In the first, construct an array of at least five elements containing numeric information. Pass to the second procedure the number of elements in the array and a pointer to the first element of the array. In the second procedure, sort the elements of the array into ascending sequence.

A solution to this problem appears on the following page.

```
/* SOLUTION TO CHECKPOINT 5 PROBLEM */

FIRST:  PROCEDURE;
        DECLARE P POINTER;
        DECLARE ARRAY (5) INITIAL (18, 5, 3, 1, 4);
        P = ADDR (ARRAY);
        NO = 5;
        CALL SECOND (NO, P);
END FIRST;

SECOND:  PROCEDURE (NO, P);
         DECLARE P POINTER;
         DECLARE TABLE (5) BASED (P);
         DO J=1 TO NO-1;
                 DO I=1 TO NO-J;
                 IF P->TABLE (I) > = P->TABLE (I+1) THEN DO;
                         SAVE = TABLE (I);
                         TABLE (I) = TABLE (I+1);
                         TABLE (I+1) = SAVE;
                         END;
                 END;
         END;
END SECOND;
```

CHAPTER 6: USING ASSEMBLER LANGUAGE INSTRUCTIONS AND CONTROL PROGRAM SERVICES

With BSL, programmers can use any of the assembler language instructions or control program services that are available to them. A single instruction or macro instruction, or any number of instructions may be included as a group or may be interspersed among BSL statements. The transition from BSL text to non-BSL text, however, must be indicated with a statement called the GENERATE statement.

There are two forms of the GENERATE statement. One form, called the simple GENERATE statement, is used to identify individual non-BSL instructions. The other form, called the block GENERATE statement, is used to indicate a number of contiguous non-BSL instructions, and does not require that individual instructions in the block be identified.

The simple GENERATE statement consists of the statement keyword GENERATE, followed, in parentheses, by one assembler language instruction or system macro instruction:

```
GENERATE ( One Assembler Language Instruction
                          -or-                      ) ;
           One System Macro Instruction
```

A Set Program Mask instruction could be placed in a BSL program as follows:

```
GENERATE  (SPM 1);
```

Similarly, a GETMAIN system macro instruction could be specified as follows:

```
GENERATE  (GETMAIN R, LV=(0));
```

The block GENERATE statement consists only of the keyword GENERATE and indicates the beginning of a sequence of non-BSL instructions. When this form is used, the special delimiter $ENDGEN must be used to indicate the end of the sequence:

```
GENERATE;

Assembler Language

Instructions and/or

System Macro Instructions

$ENDGEN
```

When simple GENERATE statements are used, the non-BSL instructions are placed into columns 10-72 of the input to the assembler, and statement names appearing on the GENERATE statements are placed into columns 1-8. On the other hand, when the block GENERATE statement is used, non-BSL instructions are placed into columns 1-80 of the input to the assembler.

Several things must be considered whenever non-BSL instructions are placed into a BSL program. These include access to data items with non-BSL instructions and also access to general registers. They require the user to have a knowledge of how the BSL compiler produces code and, therefore, are beyond the scope of this publication. Such considerations are described in the publication: BSL User's Guide.

Listed below are all BSL keywords, including both statement keywords and attribute keywords. Some have not been described in this publication, as it is intended to be only a general introduction to the language. They are shown here to provide a checklist of symbolic names that should not be used (symbolic names connot be the same as some BSL keywords). A "YES" in the column marked Reserved indicates that the corresponding keyword should not be used as a symbolic name.

Any of the abbreviations shown below may be used freely in place of the corresponding keywords.

Keyword	Permissible Abbreviation	Reserved	Keyword	Permissible Abbreviation	Reserved
ABNORMAL	ABNL		GO TO	GOTO	YES
ABS		YES	IF		YES
ADDR		YES	INITIAL	INIT	
AUTOMATIC	AUTO		INTERNAL	INT	
BASED			LABEL		
BIT			LOCAL		
BOUNDARY	BDY		NONLOCAL		
BY		YES	NORMAL		
CALL		YES	NOSAVEAREA		
CHARACTER	CHAR		POINTER	PTR	
CODEREG			PROCEDURE	PROC	YES
DATAREG			REENTRANT		
DECLARE	DCL	YES	REGISTER	REG	
DO		YES	RELEASE		YES
DONTSAVE			RESTRICT		YES
ELSE		YES	RETURN		YES
END		YES	RETURN TO		YES
ENTRY		YES	SAVE		
EXTERNAL	EXT		STATIC		
FIXED			THEN		YES
GENERATE	GEN	YES	TO		YES

APPENDIX B: PRIORITY OF OPERATORS

The following is a list of all BSL operators, and the priority of each. In complex expressions--those involving more than one operator--the operations are performed according to the priority of the operators. When more than one operator of the same priority appears in the same expression, the corresponding operations are performed in the order by which they appear in the expression; that is, from left to right.

Operator	Operation	Priority
+	Prefix Plus	1
-	Prefix Minus	1
*	Multiplication	2
/	Division (Quotient)	2
//	Division (Remainder)	2
+	Addition	3
-	Subtraction	3
>	Comparison (Greater Than)	4
<	Comparison (Less Than)	4
=	Comparison (Equal To)	4
¬ >	Comparison (Not Greater Than)	4
¬ <	Comparison (Not Less Than)	4
¬ =	Comparison (Not Equal To)	4
> =	Comparison (Greater Than or Equal To)	4
< =	Comparison (Less Than or Equal To)	4
&	And	5
|	Or	6
&&	Exclusive Or	7