Burroughs

B 7000/B 6000

ALGOL

REFERENCE MANUAL

PRICED ITEM

Printed in U.S.A. May 1977 5001639

Burroughs

B 7000/B 6000

ALGOL

REFERENCE MANUAL

PRICED ITEM

Printed in U.S.A. May 1977 5001639

LIST OF EFFECTIVE PAGES

Page No.	Issue
Title	Original
A	Original
i thru vi	Original
1-1 thru 1-2	Original
2-1 thru 2-11	Original
2-12	Blank
3-1 thru 3-3	Original
3-4	Blank
4-1 thru 4-72	Original
5-1 thru 5-120	Blank
6-1 thru 6-33	Original
6-34	Blank
A-1 thru A-5	Original
A-6	Blank
B-1 thru B-11	Original
B-12	Blank
C-1 thru C-5	Original
C-6	Blank
D-1 thru D-35	Original
D-36	Blank
E-1 thru E-7	Original
E-8	Blank
F-1 thru F-4	Original
G-1 thru G-4	Original
H-1 thru H-5	Original
H-6	Blank
Index-1 thru Index-35	Original
Index-36	Blank

Correspondence regarding this document should be addressed directly to Burroughs Corporation, P.O. Box 4040, El Monte, California 91734, Attn: Publications Department, TIO–West.

PREFACE

The purpose of this manual is to provide a reference document for the experienced programmer who is familiar with the **B 7000/B 6000** Extended **ALGOL** Language, hereinafter referred to as **ALGOL**, and the **B 7000/B 6000** Information Processing System. This reference manual is intended neither as a primer nor as a tutorial document.

ALGOL is based on the definitive "Revised Report on the Algorithmic Language **ALGOL 60**" (Communications of the **ACM**, Vol. 6, No. 1; January, 1963).

ALGOL, in addition to implementing virtually all of **ALGOL 60**, has provisions for extensive communication between programs and input-output devices, enables editing of data, and implements diagnostic mechanisms for program debugging.

This reference manual is divided into the following six sections and eight appendices:

- Section 1, **METALANGUAGE DEFINITION**: this section explains the syntactical notation used in defining the **ALGOL** language.
- Section 2, **LANGUAGE COMPONENTS**: this section describes the elements that form the most primitive structures in the language.
- Section 3, **PROGRAM STRUCTURE**: this section describes the basic structure of an **ALGOL** program.
- Section 4, **DECLARATIONS**: this section describes how elements are declared prior to being manipulated via statements and expressions.
- Section 5, **STATEMENTS**: this section presents the language element (the <*statement*>) that causes a specified action to be performed.
- Section 6, **EXPRESSIONS**: this section describes the primary active element of the language.
- Appendix A, **RESERVED WORDS**: this appendix is a list of "words" that have been set aside for specific purposes within **ALGOL**.
- Appendix B, **PROGRAM CHARACTER AND WORD FORMATS**: this appendix illustrates and describes the various characters and words **(B 7000/B 6000)** that can be used and accessed by the programmer.
- Appendix C, **CHARACTER SETS AND CODING FORM.**
- Appendix D, **COMPILE TIME OPTIONS**: this appendix describes the compiler options available to the user.
- Appendix E, **PROGRAM SOURCE AND OBJECT FILES**: this appendix describes how compiler communication is handled through various input and output files.
- Appendix F, **BATCH FACILITY**: this appendix describes the method by which batch programs can be grouped to reduce the cost of required system overhead functions.

- Appendix G, **RUN-TIME FORMAT ERROR MESSAGES**: this appendix lists and explains the error messages that occur at run-time because of formatting errors.

- Appendix H, **COMPILE-TIME FACILITIES**: this appendix describes how **ALGOL** source data can be compiled conditionally or interactively.

As an additional aid, the language elements have been arranged in alphabetical sequence within each section.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. METALANGUAGE DEFINITION

SCOPE OF THE LANGUAGE

ALGOL 60 is a language designed to represent algorithms or procedures for calculation. Extended **ALGOL**, hereinafter referred to as **ALGOL**, also includes facilities for communicating algorithms to the B 7000/B 6000 Information Processing System.

ALGOL employs a vocabulary of reserved words and symbols. These reserved words and symbols cannot be used in a program for any purpose other than defined by the language description in this manual.

Reserved words and symbols are grouped in ways prescribed by the syntax to form the various constructs of the language. These constructs can be divided into five major categories: language components, prograr unit, declarations, statements, and expressions.

Language components form the basis on which the entire **ALGOL** language is built.

A program unit is the smallest "compilable" grouping of syntactic entities. The typical **ALGOL** program is a program unit, and it contains declarations, statements, and expressions to accomplish the program's objectives.

Declarations are provided in the language for giving the **ALGOL** compiler information about the constituents of the program such as array sizes, the types of values that variables can assume, or the existence of subroutines. Each such entity must be named by an identifier and all identifiers must be declared before they are used.

Statements provide means of assigning values and results of computation, iterative mechanisms, conditional and unconditional transfers of program control, and input/output operations. In order to provide control points for transferring program control, statements can be labeled.

Expressions are rules by which values can be obtained by executing various operations on the primary elements of which expressions are composed.

SYNTAX DESCRIPTION

The syntax of the language is described in Backus-Naur form (BNF) notation. The metalinguistic symbols have the following meanings:

SYMBOL	DESCRIPTION
< >	Left and right broken brackets are used to contain one or more characters representing a metalinguistic variable whose definition is given by a metalinguistic formula.
::=	The symbol ::= means "is defined as", and separates the metalinguistic variable on the left of the formula from its definition on the right.
|	The symbol | means "or." This symbol separates alternative definitions of a metalinguistic variable.

Definition

METALANGUAGE

Continued

{ } These braces are used to enclose metalinguistic variables that are defined by the meaning of the English language expression contained within the braces. The convention is used only when it is impossible or impractical to use a metalinguistic formula.

The above metalinguistic symbols are used in forming a metalinguistic formula. A metalinguistic formula is a rule that produces a syntactically correct sequence of characters and/or symbols. The entire set of such formulae defines the constructs of **ALGOL.**

Any mark or symbol in a metalinguistic formula that is not one of the above metalinguistic symbols denotes itself. The juxtaposition of metalinguistic variables and/or symbols in a metalinguistic formula denotes juxtaposition of these elements in the construct indicated.

Spaces have been used between language elements for readability in this document, but in general, spaces cannot be used or omitted except as prescribed herein.

The metalinguistic formula below is read as follows: An *<identifier>* is defined as a *<letter>*, or an *<identifier>* followed by a *<letter>*, or an *<identifier>* followed by a *<digit>*.

```
<identifier> ::= <letter> |
                 <identifier> <letter> |
                 <identifier> <digit>
```

The metalinguistic formula above also defines a recursive relationship by which a construct called an *<identifier>* can be formed. Evaluation of the formula shows that an *<identifier>* begins with a *<letter>*, the *<letter>* can stand alone, or it can be followed by any mixture of *<letter>s* and *<digit>s.*

2. LANGUAGE COMPONENTS

LANGUAGE COMPONENTS

Syntax

<language components> ::= *<basic symbol>* |
<define invocation> |
<identifier> |
<number> |
<remark> |
<reserved word> |
<string> |
<program unit>

Semantics

<basic symbol>, *<identifier>*, *<number>*, *<remark>*, and *<string>* are discussed in this section.

The *<define invocation>* is explained under the *<define declaration>*, although the *<define invocation>* can be used anywhere in a program.

<reserved word>s are explained and listed in appendix A.

<program unit> is discussed in section 3, **PROGRAM STRUCTURE.**

BASIC SYMBOL

Syntax

```
<basic symbol> ::= <empty> |
                   <letter> |
                   <digit> |
                   <delimiter>
<empty> ::= {the null set of characters}
<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
             N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<delimiter> ::= <bracket> |
                <operator> |
                <space>
<bracket> ::= ( | ) | [ | ] | " | BEGIN | END | # | LB | RB
<operator> ::= <arithmetic operator> |
               <logical operator> |
               <relational operator> |
               := |
               &
<arithmetic operator> ::= + | * | TIMES | MUX | / | DIV | MOD | ** | –
<logical operator> ::= AND | OR | NOT | EQV | IMP | | |
<relational operator> ::= LEQ | <= |
                          LSS | < |
                          EQL | = | IS |
                          NEQ | ¬= | ISNT |
                          GTR | > |
                          GEQ | >=
<space> ::= <single space> |
            <space> <single space>
<single space> ::= {one blank position}
```

Semantics

LETTERS

Only uppercase *<letter>s* are permitted. The lowercase *<letter>s* are specifically disallowed. Individual *<letter>s* do not have particular meanings except as used in pictures and formats.

DIGITS

<digit>s are used for forming *<number>s, <identifier>s,* and *<string>s.*

DELIMITERS

Delimiters are the class of *<operator>s, <space>s, <bracket>s.* As the word "delimiter" indicates, an important function of these elements is to delimit the various entities that make up a program.

Each *<delimiter>* has a fixed meaning which, if not obvious, is explained elsewhere in this document in the syntax of appropriate constructs. Delimiters and logical values are considered *<basic symbol>s* of the language, having no relation to the individual *<letter>s* of which they are composed. Consequently, the words that constitute the *<basic symbol>s* are reserved for specific use in the language. A complete list of these words, called *<reserved words>s,* and details of the applicable restrictions are given in appendix A.

SPACE

IN **ALGOL 60**, spaces have no significance since basic components of the language such as **BEGIN** are construed as one symbol. However, in a machine implementation of such a language, this approach is not convenient for the **ALGOL** programmer. In **ALGOL**, for instance, **BEGIN** is composed of five letters, **TRUE** is composed of four, and **PROCEDURE** of nine. No *<space>* can appear between the letters of a *<reserved word>*; otherwise, it is interpreted as two or more elements.

The *<reserved word>s* and *<basic symbol>s* are used, together with *<variable>s* and *<number>s,* to form *<expression>s, <statement>s,* and *<declaration>s.* Because some of these constructs place quantities that have been defined by the programmer next to *<delimiter>s* composed of *<letter>s* it is necessary to separate one from the other. The *<space>* is used as a delimiter in these cases. Therefore, a *<space>* must separate any two basic *<language component>s* of the following forms:

a. Multicharacter delimiter (except :=, **, @@,¬=, ⟨=, ⟩=,).
b. Identifier.
c. Logical value.
d. Unsigned number.

Aside from these requirements, a *<space>* can appear, if desired, between any two *<basic component>*s without affecting their meaning.

IDENTIFIER

Syntax

<identifier> ::= *<letter>* |
<identifier> <letter> |
<identifier> <digit>

Semantics

<identifier>s have no intrinsic meaning. They name labels, variables, arrays, procedures, etc. An *<identifier>* can be no more than 63 *<character>s* long and cannot include *<space>s* or *<visible special character>*. An identifier must start with a letter, which can be followed by any combination of letters or digits, or both. The same *<identifier>* can be used to denote two different entities only when the "scopes" of these entities do not overlap. The multi-character symbols for relational and logical operators can be declared as identifiers. However, if declared, they cannot be used as operators within the scope of the declaration. Examples of legal and illegal identifiers are as follows:

LEGAL IDENTIFIERS	ILLEGAL IDENTIFIERS
A	BEGIN
I	1776
B5	2BAD
YSQUARE	$
TOOBAD	X-Y
LONGTONS	W-2 FORM
LAZY8	<CAPTION>
PRESSURE	SEC(X)
XOVERZ	RATE-HR
D2P471GL	NO.

NUMBER

Syntax

```
<number> ::= <sign> <unsigned number>
<sign> ::= <empty> |
           + |
           -

<unsigned number> ::= <decimal number> |
                      <exponent part> |
                      <decimal number> <exponent part>
<decimal number> ::= <unsigned integer> |
                     <decimal fraction> |
                     <unsigned integer> <decimal fraction> |
                     <unsigned integer> .
<unsigned integer> ::= <digit> |
                       <unsigned integer> <digit>
<decimal fraction> ::= . <unsigned integer>
<exponent part> ::= @<integer> |
                    @@<integer>
<integer> ::= <sign> <unsigned integer>
```

Examples

UNSIGNED INTEGERS	DECIMAL FRACTIONS	DECIMAL NUMBERS
5	.5	.69
69	.69	.546
	.013	3.98

INTEGERS	EXPONENT PARTS	UNSIGNED NUMBERS
1776	@8	99.44
-62256	@-06	@-11
548	@+54	1354.543@48
		.1864@4

NUMBERS	ILLEGAL NUMBERS
0	50 00.
+549755813887	1,505,278.00
1.75@-46	@63.4
-4.31468	5@8 8
-@2	1@2.5
.375	1.667E-01

Semantics

Numbers can be of two basic types, integer or real.

No <*space*> can appear within an <*unsigned integer*>. All numbers that do not contain the exponent <*delimiter*> (@@) are considered to be single-precision.

The illegal number examples, given above, emphasize the fact that the only characters which are used to form numbers are <*digit*>*s* and the <*basic symbol*>*s* ., @, +, and -. Note that no provision is made for <*space*>*s* to occur inside <*number*>*s*.

NUMBER RANGES

The largest and smallest integers and numbers that can be represented are as follows (decimal versions are approximate):

a. Any integer between and including plus and minus 549755813887 = 8**13-1 = 4"007FFFFFFFFF" can be represented in integer form.
b. The largest positive normalized single-precision number is 4.31359146674@68 = (8**13-1) 8*8863 = 4"1FFFFFFFFFFF".
c. The smallest positive normalized single-precision number is 8.75811540203@-47 = 8**(-51) = 4"3F9000000000". The number zero and negative numbers with absolute value between the largest and smallest values given above may be represented in real form.
d. The largest positive normalized double-precision number is 1.948829382050280791244469@@ 29603 = (8**26-1) *8**32767 = 4"1FFFFFFFFFFFFFFFFFFFFFFF".
e. The smallest positive normalized double-precision number is 1.93854585713758583355 64@@-29581 = 8**(-32742) = 4"3F9000000000FF8000000000". The number zero and positive and negative numbers with absolute value between the largest and smallest values given above may be represented in double form.

COMPILER NUMBER CONVERSION

The **ALGOL** compiler can convert a maximum of 24 significant decimal digits of mantissa in double-precision. The "effective exponent", which is the explicit exponent value following the @@ sign minus the number of digits to the right of the decimal point, must be less than 29604 in absolute value. For example, the final fractional zero (0) cannot be specified in the smallest positive normalized double-precision number shown above: -29581-(23 fractional digits) = -29604. Leading zeros are not counted in determining the number of significant digits. For example, 0.0002 has one significant digit, whereas 1.002 has five significant digits.

SYMMETRY OF THE NUMBER SETS

The number sets are symmetrical with respect to zero, that is, the negative <*number*> corresponding to any valid positive <*number*> can also be expressed in the language and the object program.

EXPONENTS

The exponent part is a scale factor expressed as an integral power of 10. The exponent part @@ <*integer*> signifies that the entire number is a double-precision value.

REMARK

Syntax

<remark> ::= *<end remark>* |
<comment remark> |
<escape remark>
<end remark> ::= {any sequence of *<letter>s*, and *<digit>s*, and *<space>s* not containing the reserved word **END**, **ELSE**, or **UNTIL**}
<comment remark> ::= **COMMENT** {any sequence of **EBCDIC** characters not containing a semicolon};
<escape remark> ::= % {any sequence of **EBCDIC** characters extending to the end of the logical source record }

Examples

The following program illustrates the syntactically correct uses of the **COMMENT**.

```
BEGIN
FILE F(KIND = PRINTER COMMENT;);
FORMAT COMMENT; FMT COMMENT; (A4, I6);
PROCEDURE P (X,COMMENT; Y,Z);
REAL X,Y COMMENT; , Z ; % PERCENT SIGN CAN BE USED HERE
X := Y + COMMENT; Z; % HERE TWO
IF COMMENT; 7 〉 5 THEN WRITE (F,〈"OK"〉);
IF 4 COMMENT; 〉2 THEN WRITE(F, 〈"OK"〉);
IF 8 〉 5 THEN WRITE COMMENT; (F, 〈"OK"〉);
END OF PROGRAM.
```

The following program illustrates the improper use of the **COMMENT**.

```
BEGIN
FILE F(KIND=PRINTER);
FORMAT FMT(13,F10.3 COMMENT; , A4);
ARRAY A[0:99];
REAL X;
FORMAT ("ABC", % CANNOT BE USED. "DEE");
WRITE(F,〈"INVALID USE" COMMENT; 〉);
REPLACE POINTER(A) BY "ABCD COMMENT; EFGHIJ";
X := "AB,COMMENT; C";
COMMENT CANNOT BE USED HERE COMMENT; EITHER;
END.
```

Semantics

Three methods are provided in the language to insert program documentation at various locations throughout the source file. The *<end remark>* is allowed following the particular *<basic component>*, **END**. The compiler recognizes the termination of the *<end remark>* upon finding the reserved word **END**, **ELSE**, or **UNTIL**, or upon finding a non-alpha, non-numeric **EBCDIC** character.

The *<comment remark>* is allowed between any two *<basic component>s,* except within an

<editing specification> (refer to the *<format declaration>*) and except following the *<editing specification>s* of a *<format declaration>* but prior to the end of the same *<format declaration>*. The compiler considers the first semicolon encountered after the reserved word **COMMENT** as the end of the *<comment remark>*. Note that since a *<remark>*, a *<string>*, and an *<invocation>* are each *<basic component>s*, a *<comment remark>* is not recognized within a *<string>*, an *<invocation>* or another *<remark>*. Also note that *<string>s*, and *<escape remark>s* can each contain the dollar character ($). Care must be exercised in the case of the *<string>* or the *<comment remark>* to ensure that these constructs do not contain a dollar character in position 1 of the source record or a blank character followed by a dollar character in positions 1 and 2 of the source record. An error in this respect causes the compiler to interpret the source record as a compiler control record. The structure of the *<escape remark>* cannot lead to this error.

The percent sign (%) starting an *<escape remark>* must follow a *<basic component>* not contained in an *<editing specification>s.* The *<escape remark>* extends from the starting percent sign and extends to the start of the sequence number field. The compiler does not examine the *<escape remark>*. When the percent sign is encountered that starts an *<escape remark>*, the compiler skips immediately to the next source record before continuing the compilation process.

STRING

Syntax

```
<string> ::= <simple string> |
             <string> <simple string>
<simple string> ::= <numeric string> |
                    <alpha string>
<numeric string> ::= <binary code> " <binary string> " |
                     <quaternary code> " <quaternary string> " |
                     <octal code> " <octal string> " |
                     <hexadecimal code> " <hexadecimal string> "
<binary code> ::= 1 | 10 | 12 | 120 | 13 | 130 | 14 | 140 | 16 | 160 |
                  17 | 170 | 18 | 180
<binary string> ::= <binary character> |
                    <binary string> <binary character>
<binary character> ::= 0 | 1
<quaternary code> ::= 2 | 20 | 24 | 240 | 26 | 260 | 27 | 270 | 28 | 280
<quaternary string> ::= <quaternary character> |
                        <quaternary string> <quaternary character>
<quaternary character> ::= 0 | 1 | 2 | 3
<octal code> ::= 3 | 30 | 36 | 360
<octal string> ::= <octal character> |
                   <octal string> <octal character>
<octal character> ::= 0 | 1 | 2 | 3 | 4 | 5 | 7
<hexadecimal code> ::= 4 | 40 | 47 | 470 | 48 | 480
<hexadecimal string> ::= <hexadecimal character> |
                         <hexadecimal string> <hexadecimal character>
<hexadecimal character> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B |
                            C | D | E | F
<alpha string> ::= <EBCDIC code> " <EBCDIC string> " |
                   <BCL code> " <BCL string> " |
                   <ASCII code> " <ASCII string> "
<EBCDIC code> ::= <empty> | 8 | 80
<EBCDIC string> ::= " |
                    <EBCDIC character> |
                    <EBCDIC string> <EBCDIC character>
<EBCDIC character> ::= <letter> |
                       <digit> |
                       <visible special character>
<visible special character> ::= . | , | [ | ] | ( | ) | + | – | / | 〉 | 〈 |
                                = | % | & | * | # | @ | : | ; | $ | "
<BCL code> :: = 6 | 60
<BCL string> ::= <EBCDIC string>
<ASCII code> ::= 7 | 70
<ASCII string> ::= <EBCDIC string>
```

Semantics

CHARACTER SIZE

Strings can be composed of binary (1-bit), quaternary (2-bit), octal (3-bit), hexadecimal (4-bit), **BCL** (6-bit), **ASCII** (7-bit in 8-bit format), and **EBCDIC** (8-bit) characters.

STRING CODE

The string code determines the interpretation of the characters between the quotes. It specifies the character set and, for strings of less than 48 bits, the justification. The first digit of the string code specifies the character set in which the source string is written. The next non-zero digit (if any) specifies the internal character size of the string to be created by the compiler. If no next non-zero digit is specified, the internal size is the same as the source size. If the internal size is different from the source size, the length of the string must be an integral number of internal characters.

If the <*string*> contains fewer than 48 bits, a trailing zero in the string code specifies that the <*string*> is to be left-justified within a word and that trailing zeroes are to fill out the remainder of the word.

If the <*string*> contains fewer than 48 bits, the absence of a trailing zero in the string code specifies that the <*string*> is to be right-justified within a word and that leading zeroes are to fill out the remainder of the word.

If the <*string*> contains 48 or more bits, the presence or absence of a trailing zero in the string code is irrelevant.

If the string code is <*empty*>, its default value is 8 or 6 depending on the "default character size" of the program. Note that 8-bit (**EBCDIC**) is assumed by the **ALGOL** compiler unless the programmer uses a **BCL** compiler control card, which indicates 6-bit.

ASCII CHARACTERS

An <**ASCII** *code*> can be used only with <**ASCII** *string*>*s* that contain only characters having corresponding **EBCDIC** graphics, since these are the only characters recognized by the compiler.

The compiler translates each **ASCII** character into an 8-bit character, the rightmost seven bits of which are the **ASCII** representation of that graphic and the leftmost bit is a zero.

For characters that are not in the **EBCDIC** character set, the **ASCII** characters must be written as a <*hexadecimal string*>, where each pair of hexadecimal characters represents the internal code of one **ASCII** character, right-justified with a leading zero bit.

QUOTE CHARACTER

The quote character (") can appear only as the first <*character*> of a <*simple string*>. Strings with internal quotes must be broken into separate simple strings by the use of three quotes in succession. Notice the syntax for <**EBCDIC** *string*>.

STRING LENGTH

The maximum permissible string length depends upon the context in which the *<string>* is used. List elements and fill statements that consist only of a string can include strings up to 256 8-bit characters in length. Strings used as primaries in arithmetic expressions are limited to a length of 48 bits.

A *<string>* less than 49 bits is represented internally as an 8-bit, 16-bit, or 48-bit literal (depending on the number of bits in the *<string>*) in the code segment. A *<string>* longer than 48 bits is carried in a "pool array" of the object program, and is referenced by a pointer whose character size is appropriate for the string.

All string parameters are terminated with at least one null character. If the string ends on a word boundary, an extra word is added to the array to contain the null. Thus, an **ALGOL** program can find the end of the string by scanning WHILE 〉 48"00".

COMPOSITE STRINGS

When a *<string>* is formed from simple strings of different character sizes, the following rules apply:

a. The justification specified by each string code after the first is ignored.
b. Every *<character>* must be aligned at a character boundary appropriate for that character size. For example, 3 "6" 6 "8" results in an error and must be replaced by 3 "6" 3 "0" 6 "8", or 3 "06" 6 "8", or 3 "60" 6 "8".
c. The maximum character size must be divisible by all character sizes that appear in the *<string>*. Thus, it is impossible to mix 6-bit characters with 4-bit characters. For example, 3 "7" 2 "1" is illegal. Note that 7-bit characters are really 8-bit characters with one unused bit.

3. PROGRAM STRUCTURE

PROGRAM UNIT

Syntax

<program unit> ::= *<block>*. |
<compound statement>. |
<global part> *<procedure declaration>*. |
<global part> *<procedure declaration>*; |
<block> ::= *<block head>* ; *<compound tail>*
<block head> ::= **BEGIN** *<declaration>* |
<block head> ; *<declaration>*
<compound tail> ::= *<statement>* **END** |
<statement> ; *<compound tail>*
<compound statement> ::= **BEGIN** *<compound tail>*
<global part> ::= *<empty>* |
[*<declaration list>*]
<declaration list> ::= *<declaration>* |
<declaration list> ; *<declaration>*

Examples

```
BEGIN REAL X; END.
BEGIN END.
PROCEDURE P; BEGIN END.
REAL PROCEDURE Q; BEGIN Q  := 4 END.
PROCEDURE S; BEGIN REAL X; END;
[REAL S; ARRAY B[1]; FILE LINE; PROCEDURE RD(V); VALUE V; REAL V; EXTERNAL;]
```

Semantics

A *<block>* is a *<statement>* that groups one or more *<declaration>s* and *<statement>s* into a logical entity. A *<compound statement>* is a *<statement>* that has no *<declaration>s* following the **BEGIN** of a **BEGIN-END** pair; that is, a *<compound statement>* provides the means of grouping several *<statement>s* into some form of logical unit.

A *<compound statement>*, and a *<block>* are recursive in that their definitions involve *<statement>s* and/or *<declaration>s*. A *<statement>* can itself be a *<statement>*. The structure of *<compound statement>s* and *<block>s* are illustrated as follows (**S** represents any *<statement>* and **D** represents any *<declaration>*):

<compound statement>s

```
BEGIN S;S;S;...;S; S END
BEGIN S;S;BEGIN S;BEGIN S;S END END; S END
```

<block>s

```
BEGIN D;D;...;D;S;S;S;...;S;S; END
BEGIN D;D;S;BEGIN D;S END;BEGIN S;S;S END END
```

A *<program>* that has the form of a *<procedure declaration>* is typically used as a unit that is bound to a more complete program.

A *<global part>* allows global entities to be referenced by a separate *<procedure declaration>*. The global entities must be declared prior to the *<procedure declaration>* itself. Any *<program unit>* that has a *<global part>* is valid only for binding to a host.

Pragmatics

A *<compound statement>* is executed in-line and does not change the memory requirement of a program. A *<block>* must be entered by the procedure entry operator and does modify the memory requirement of a *<program unit>*. (Entering a *<block>* costs CPU resources; entering a *<compound statement>* does not cost anything extra.)

A *<program unit>* can be preceded by *<remark>*; it cannot be followed by a *<remark>*.

SCOPE

Those portions of an **ALGOL** program where an *<identifier>* can be used to successfully reference its corresponding entity are defined to be the scope of that entity.

In one part of an **ALGOL** program, an *<identifier>* can reference an entity; and in another part of the program, the same *<identifier>* can reference a different entity. The scope of each entity is defined in such a way that at any point in the program, an *<identifier>* references at most one entity.

The scope of an entity is described by rules that define which parts of the program are included in the scope, which parts of the program are excluded by the scope, and the uniqueness requirements placed upon the choice of identifiers. Those general rules are described in the following paragraphs.

Local Entities

An *<identifier>* that is declared within a *<block>* is referred to as being "local" to that *<block>*. The entity associated with that *<identifier>* inside the *<block>* is not associated with that *<identifier>* outside the *<block>*. In other words, on entry to a *<block>*, the values of local entities are undefined; on exit from the *<block>*, the values of local entities are lost, but those of "global" entities (refer to **GLOBAL ENTITIES**) are retained. However, the *<identifier>* associated with some entity may be referred to inside inner *<block>s*, where it becomes a global *<identifier>* relative to the inner *<block>s*.

Global Entities

An identifier that appears within a *<block>* and is not declared within that *<block>* but is declared in an outer *<block>* is referred to as being "global" to that *<block>*. Therefore, a global *<identifier>* represents the same entity inside the *<block>* and outside of that *<block>*.

As the following program illustrates, an *<identifier>* can be local relative to one reference and global relative to another reference.

```
BEGIN
FILE F (KIND = PRINTER);
REAL A;
A := 7; % FIRST STATEMENT OF OUTER BLOCK
  BEGIN
  LIST L1 (A);
  INTEGER A;
  LIST L2 (A);
  A := 3; % FIRST STATEMENT OF INNER BLOCK
  WRITE (F, <3R10.2>, L1);
  WRITE (F, <3R10.2>, L2);
  END OF INNER BLOCK
END OF PROGRAM.
```

In the preceding example, the *<identifier>*, **A**, appears in *<list>* **L1** as a global reference and references the **REAL A.** In *<list>* **L2**, the identifier, **A**, references the **INTEGER A.**

Global entities can be used in inner blocks in the following ways:

a. To carry values into the *<block>* values that have been calculated in an outer block.
b. To carry into an outer *<block>* a value calculated inside the *<block>*.
c. To preserve a value calculated within a *<block>* for use in a later entry to the same *<block>*.
d. To transmit a value from one *<block>* to another *<block>* neither containing nor contained within the first *<block>*.

4. DECLARATIONS

DECLARATION

Syntax

<declaration> ::= *<array declaration>* |
<array reference declaration> |
<define declaration> |
<direct array declaration> |
<dump declaration> |
<event declaration> |
<event array declaration> |
<file declaration> |
<format declaration> |
<forward reference declaration> |
<interrupt declaration> |
<label declaration> |
<list declaration> |
<monitor declaration> |
<picture declaration> |
<pointer declaration> |
<procedure declaration> |
<switch declaration> |
<task declaration> |
<task array declaration> |
<translatetable declaration> |
<truthset declaration> |
<type declaration> |
<value array declaration>

Semantics

A *<declaration>* defines certain properties of entities and relates these entities to *<identifier>s*. Every *<identifier>* must be "declared" prior to using it in an ALGOL program. The compiler ensures that subsequent usage of the *<identifier>* in the program is consistent with its declaration.

Declarations
ALPHA

ALPHA DECLARATION

Syntax

<alpha declaration> ::= *<local or own>* **ALPHA** *<identifier list>*
<local or own> ::= *<empty>* |
OWN
<identifier list> ::= *<identifier>* |
<identifier list> , *<identifier>*

Examples

```
ALPHA ALFA
ALPHA BETA, GAMMA, CHARS, ACCUM
OWN ALPHA MYPERSONALALPHA
```

Semantics

An *<alpha declaration>* is used to declare *<simple variable>s* which can be used as alphanumeric values. Character values are either six, 8-bit characters (normal) or eight, 6-bit characters (**BCL**).

The *<local or own>* portion of the *<alpha declaration>* indicates whether the value of the specified *<simple variable>* is to be retained upon exit from the *<block>* in which it is declared. A *<simple variable>* declared to be **OWN** will retain its value when the program exits from the associated *<block>*, and that "old" value will be the contents of the *<simple variable>* when the associated *<block>* is re-entered.

Upon entry to a *<block>* containing *<simple variable>s,* the normal content of a non-**OWN** *<simple variable>* is a zero (0); i.e., a 48-bit word with all bits off. To be truly compatible with **ALGOL-60**, a programmer would explicitly zero the *<simple variable>s* with appropriate *<assignment statement>s.*

Pragmatics

A *<simple variable>* declared as **ALPHA** is treated as a **REAL** in terms of storing and manipulation.

Appendix B contains additional information on the internal structure of an alpha *<simple variable>* as implemented on the **B 7000/B 6000** Information Processing System.

ARRAY DECLARATION

Syntax

<array declaration> ::= *<long/own specification>* *<array class>* **ARRAY** *<array list>*
<long/own specification> ::= *<local or own>* |
LONG *<local or own>*
<array class> ::= *<empty>* |
<type> |
<character type>
<type> ::= **ALPHA** |
BOOLEAN |
DOUBLE |
INTEGER |
REAL
<character type> ::= **HEX** |
BCL |
ASCII |
EBCDIC
<array list> ::= *<array segment>* |
<array list> , *<array segment>*
<array segment> ::= *<array identifier list>* [*<bound pair list>*] |
<array equivalence>
<array identifier list> ::= *<array identifier>* |
<array identifier list> , *<array identifier>*
<array identifier> ::= *<identifier>*
<bound pair list> ::= *<bound pair>* |
<bound pair list> , *<bound pair>*
<bound pair> ::= *<lower bound>* : *<upper bound>*
<lower bound> ::= *<arithmetic expression>*
<upper bound> ::= *<arithmetic expression>*
<array row equivalence> ::= *<array identifier>* [*<lower bound>*] = *<array row>*

Examples

```
INTEGER ARRAY MATRIX [1:IF B2 THEN B + K ELSE B + I]
INTEGER ARRAY DOG [0:5, 0:25, 1:7, 4:16]
OWN REAL ARRAY GROUP [0:9]
OWN BOOLEAN ARRAY GATE [1:10, 3:9]
REAL ARRAY STUB [0:4, 1:6], CAD[400:500, 1:50]
ARRAY XRAY [X+Y+Z:3*A+B];
EBCDIC ARRAY GROUPE[0] = GROUP [*]
ARRAY PARTARAY [7] = MAJORARAY
LONG ARRAY BIGY [0:9999]
ARRAY SEGARAY [0:50000]
```

Semantics

An *<array declaration>* declares one or more identifiers to represent arrays of fixed but arbitrary dimensions. Particular elements in an array are referenced by using the *<array identifier>* with a *<subscript list>* in the form of a *<subscripted variable>*.

LONG ARRAYS

The **LONG** declarator affects one-dimensional arrays only. Normally a one-dimensional array greater than 1023 words (detected at compile-time) is automatically segmented at run-time into one or more rows of 256 words each. The **LONG** declarator is used to override this segmentation. The "long" control card option may also override segmentation at run time if the array is declared to be longer than 1023 48-bit words. (Refer to **B 6800** System Operation Guide Reference Manual, form 5001563.)

The subscript bounds for an array are given in the first *<bound pair list>* following the *<array identifier>*.

<bound pair list>

The *<bound pair list>* gives the lower and upper bounds of all subscripts taken in order from left-to-right. The dimensions of the array equal the number of bound pairs in the *<bound pair list>*.

ARRAY *<type>s*

All arrays declared in an *<array declaration>* are of the same *<type>*. If the *<type>* is omitted, it becomes type **REAL** by default.

EXPRESSIONS USED AS BOUNDS

Expressions used as array dimension bounds are evaluated once, from left-to-right, upon entering the *<block>* in which the array is declared. These expressions can depend only on values that are global to that *<block>* or passed in as actual parameters.

<local or own>

If an array is declared to be **OWN**, it indicates that the array and its contents are to be retained upon exit from the *<block>* in which it is declared, and therefore available upon subsequent re-entry into the *<block>*. No **OWN** arrays with variable dimensions are allowed.

Arrays not declared as **OWN** are completely re-established upon every entry into the *<block>* in which they are declared. They are also completely deallocated upon exit from the *<block>* in which they are declared.

<character type>s

A **HEX** array references data by means of a 4-bit string descriptor; a **BCL** array with a 6-bit string descriptor; and **ASCII** and **EBCDIC** arrays with 8-bit string descriptors.

<array row equivalence>

The *<array row equivalence>* can be used to establish a "copy descriptor" of the *<array row>* with a different lower bound and/or character size.

Arrays not specified as *<character type>* are "word arrays"; i.e., each element is a 48-bit word. Note that a **DOUBLE** array has two 48-bit words for each element. Word and character arrays may be passed as parameters and used as *<array row>s*. In addition, character arrays may be used as *<simple pointer expressions>*.

Restrictions

Arrays declared in the outermost *<block>* must use constant or constant expression bounds.

In all cases, upper bounds must not be less than their associated lower bounds.

Arrays cannot have more than 16 dimensions.

If an array with variable bounds is declared **OWN**:

a. Once the array is established, the bounds cannot be changed, and,
b. The values of the subscripts used in a *<subscripted variable>* referencing the array are valid only if within the established bounds.

The maximum value of a *<lower bound>* is 131,071; the minimum value of a *<lower bound>* is -131,071.

The maximum length of an array is 2^{20} - 1.

When using the **LONG** declarator, the maximum array size is determined by the overlay size at cold-start time.

The *<array identifier>* may be declared **DIRECT**. If so, then only **DIRECT** *<array designator>s* may be assigned to it. However, non-**DIRECT** *<array identifier>s* may be assigned either **DIRECT** or non-**DIRECT** *<array identifier>s*.

ARRAY REFERENCE

ARRAY REFERENCE DECLARATION

Syntax

<array reference declaration> ::= *<direct specifier>* *<array class>*
ARRAY REFERENCE *<array reference list>*
<direct specifier> ::= *<empty>* |
DIRECT
<array reference list> ::= *<array reference segment>* |
<array reference list> , *<array reference segment>*
<array reference segment> ::= *<array reference identifier list>* [*<integer lower bound list>*]

<array reference identifier list> ::= *<array reference identifier>* |
<array reference identifier list> , *<array reference identifier>*

<array reference identifier> ::= *<identifier>*
<integer lower bound list> ::= *<integer>* |
<integer lower bound list> , *<integer>*

Examples

ARRAY REFERENCE REFARRAY [3]
DIRECT ARRAY REFERENCE DIRREFARRAY [N] % N IS A DEFINED CONSTANT
EBCDIC ARRAY REFERENCE EBCDICREFARRAY [0]
DOUBLE ARRAY REFERENCE DOUBLEREFARRAY [1, 2, 3, 7]

Semantics

An *<array reference declaration>* is used to establish an *<array reference variable>*, whose purpose is to contain a "copy descriptor" of a genuine array. An *<array reference variable>* is initialized via the *<assignment statement>* form of: *<array reference variable>* := *<array designator>*. Any subsequent use of the *<array reference identifier>* behaves like a reference to the *<array designator>*. The lex level of the *<array designator>* (i.e., the level at which the array is declared) may not be greater than that of the *<array reference variable>*; in other words, the *<array reference variable>* may not be global to the *<array designator>*.

The *<array reference variable>* may be declared **DIRECT.** If so, then only **DIRECT** *<array designator>s* may be assigned to it. However, non-**DIRECT** *<array reference variable>s* may be assigned either **DIRECT** or non-**DIRECT** *<array designator>s.*

If *<array class>* is *<empty>*, **REAL** is assumed. If the number of dimensions of the *<array reference variable>* and *<array designator>* is greater than one (single), their *<array class>* must agree. If they are single-dimensioned, the *<array class>* of the *<array designator>* may be of any *<type>*; the generated copy descriptor is modified as necessary to agree with the *<array class>* specified for the *<array reference variable>.*

The number of dimensions of the *<array reference variable>* is determined by the number of lower bounds in its declaration. There can be no more than 16 dimensions.

BOOLEAN DECLARATION

Syntax

<Boolean declaration> ::= *<local or own>* **BOOLEAN** *<identifier list>* |
BOOLEAN *<equation list>*
<equation list> ::= *<identifier list>* |
<identifier> = *<identifier>* |
<equation list> , *<identifier>* = *<identifier>* |
<equation list> , *<identifier list>*

Examples

BOOLEAN BOOL
BOOLEAN DONE, ENDOFIT, ISHOULD, TOOLATE
BOOLEAN FLAG, BINT = INTGR, ALLDONE

Semantics

A *<Boolean declaration>* is used to declare *<simple variable>s* which have a logical value of **TRUE** or **FALSE.**

The *<local or own>* portion of the *<Boolean declaration>* indicates whether the value of the specified *<simple variable>* is to be retained upon exit from the *<block>* in which it is declared. A *<simple variable>* declared to be **OWN** will retain its value when the program exits from the associated *<block>*, and that "old" value will be the contents of the *<simple variable>* when the associated *<block>* is re-entered.

Upon entry to a *<block>* containing *<simple variable>s*, the normal content of a non-**OWN** Boolean *<simple variable>* is initialized to **FALSE**; i.e., a 48-bit word with all bits off. To be truly compatible with **ALGOL-60,** a programmer would explicitly zero the *<simple variable>s* with appropriate *<assignment statement>s.*

The *<equation list>* allows address equation among real, integer, and Boolean variables only. An *<identifier>* may only be address equated to a previously declared local *<identifier>* or to an *<identifier>* global to the block in which it is declared.

Pragmatics

The **TRUE** or **FALSE** condition of the *<simple variable>* is dependent on the low-order bit (bit 0) of the word. Use of the *<partial word part>* allows all 48 bits to be tested and/or manipulated as needed.

Appendix B contains additional information on the internal structure of a Boolean *<simple variable>* as implemented on the **B 7000/B 6000** Information Processing System.

DEFINE DECLARATION and DEFINE INVOCATION

Syntax

<define declaration> ::= **DEFINE** *<definition list>*
<definition list> ::= *<definition>* |
<definition list> , *<definition>*
<definition> ::= *<defined identifier>* *<formal symbol part>* = *<text>* #
<defined identifier> ::= *<identifier>*
<formal symbol part> ::= *<empty>* |
(*<formal symbol list>*)
<formal symbol list> ::= *<formal symbol>* |
<formal symbol list> , *<formal symbol>*
<formal symbol> ::= *<identifier>*
<text> ::= { any sequence of valid characters not including a free # }
<define invocation> ::= *<defined identifier>* *<actual text part>*
<actual text part> ::= *<empty>* |
(*<closed text list>*) |
[*<closed text list>*]
<closed text list> ::= *<closed text>* |
<closed text list> , *<closed text>*
<closed text> ::= { an actual text not containing unmatched bracketing symbols or unbracketed commas }

Examples

```
DEFINE P = POINTER #
DEFINE BLANKIT = REPLACE P(LINEOUT) BY "  "  FOR 22 WORDS #
DEFINE XROOT=(-B + SQRT(B*B -4*A*C) ) / (2*A) #
DEFINE INT=INTEGRATE (X, Y,Z) #
DEFINE LP= (#, RP=) #, LEFTCHAR = [47:8] #
DEFINE FORI = FOR I := 1 STEP UNTIL #
DEFINE  FORJ(A,B,C) = FOR J := A STEP B UNTIL C #
DEFINE TAX(X) = SIN(X) / COS(X) #
DEFINE MAXX(A1,A2) = IF A1>A2 THEN A1 ELSE A2 #
DEFINE D1(X) = [31:8] #, D2(Y) = F4[X,Y] #
DEFINE DOIT(A,B) = W*A + Y.B #
DEFINE D(X,Y,Z) = X Y Z #
```

Semantics

The *<define declaration>* causes the **ALGOL** compiler to save off the specified *<text>* until such time as the associated *<define identifier>* is encountered as a *<define invocation>*. At that point, the saved off *<text>* is retrieved and compiled as if the entire *<text>* were actually located at the position of the *<define invocation>*.

A *<definition>* has two forms of syntax: (1), the "simple" define, and (2), the "parametric" define. They are readily differentiated because the parametric define has a series of parameters (or *<formal symbol>s)* enclosed in parentheses. The first six examples above are simple defines, and the last six examples are parametric defines.

The *<formal symbol>s* are an essential part of a parametric define. References to the *<formal symbol>s* cannot appear outside of the *<text>* of the corresponding parametric define.

<formal symbol>s function in a manner similar to *<formal parameters>* of the *<procedure declaration>*.

Wherever the *<formal symbol>s* appear in the *<text>*, a substitution of the corresponding *<closed text>* is made before compiling that part of the *<text>*.

The *<text>* is bracketed on the left by an equal sign (=) and on the right by a pound sign (#). This equal sign is said to be "matched" with the pound sign. The *<text>* can be any sequence of characters not containing a "free" pound sign. A free pound sign is any pound sign that is not in a *<string>*, not in a *<remark>*, and not "matched" with an equal sign in a *<define declaration>* that is nested in the *<text>*. The compiler interprets the first free pound sign as signaling the end of the *<text>*. That is, the first free pound sign is "matched" by the compiler with the equal sign that started the *<text>*.

A *<define invocation>* causes the occurrence of the *<defined indentifier>* to be replaced by the *<text>* associated with the *<define identifier>*. However, a *<define invocation>* may not appear in the *<format part list>* of a *<format declaration>* nor in the *<editing specification>s* of a *<read statement>* or *<write statement>*. Furthermore, if a *<format declaration>* or *<editing specification>* is located within the *<text>* of a parametric define, it may not reference the *<formal symbol>s* of that define. In other words, formats and defines are incompatible for invocation purposes.

The invocation of a parametric define causes textual substitution of the *<closed text>* into the indicated position(s) of the associated *<text>*. A *<closed text>* need not be "simple"; for instance, in the first of the parametric define examples above, the invocation of FORJ could be:

FORJ (0, B*3, MAX(X, Y, Z))

which, if "expanded" would be:

FOR J := 0 STEP B*3 UNTIL MAX(X, Y, Z)

Pragmatics

If the **ALGOL** compiler encounters some type of syntax error while compiling the combination of the *<text>(s)*, *<closed text>(s)*, and *<formal symbol>(s)* at the occurrence of the *<define invocation>*, the appropriate error is indicated along with a printout of the expanded define.

A maximum of nine parameters are allowed in a parametric define.

To avoid problems with expanding a define, particularly when an *<arithmetic expression>* is "passed in", each occurrence of a *<formal symbol>* in the *<text>* of a parametric define should be enclosed between parentheses. For example, **DEFINE FORJ (A, B, C) = FOR J := (A) STEP (B) UNTIL (C) #.**

Beware of passing an updating expression to a parametric define. Multiple use of the corresponding formal symbol will cause multiple updates. For example, if **DEFINE Q(E) = E + 2*E#** and **Q(X:=X+1)** is invoked, its define expands into **X:=X+1 + 2*X:=X+1;**

A syntax error will be generated when a define of a string is concatenated.

DIRECT ARRAY DECLARATION

Syntax

<direct array declaration> ::= **DIRECT** *<local or own>* *<array class>* **ARRAY** *<direct array list>*

<direct array list> ::= *<direct array segment>* |
<direct array list> , *<direct array segment>*

<direct array segment> ::= *<direct array identifier list>* [*<bound pair list>*] |
<direct array equivalence>
<direct array identifier list> ::= *<direct array identifier>* |
<direct array identifier list> , *<direct array identifier>*
<direct array identifier> ::= *<identifier>*
<direct array equivalence> ::= *<direct array identifier>* [*<lower bound>*] =
<single-dimension direct array>
<single-dimension direct array> ::= *<direct array identifier>* |
<direct array identifier> [*]

Examples

```
DIRECT ARRAY DIRARY [0:29]
DIRECT OWN REAL ARRAY MYDIRREELARAY[0:N]
DIRECT ARRAY DIREQVARAY[5] = DIRARY
DIRECT EBCDIC ARRAY MULTIDIREBCARAY[0:4, 0:20]
```

Semantics

A *<direct array declaration>* is required in order to perform Direct I/O. As stated under *<I/O statement>*, Direct I/O is handled in such a manner as to avoid use of the normal I/O facilities of the system. The primary item involved is a Direct array.

A Direct array may be word oriented or character oriented.

Direct arrays may be utilized in every way that a non-Direct can be used.

A Direct array has certain *<arithmetic-valued direct array attribute>s* and *<Boolean-valued direct array attribute>s* which can be programmatically interrogated and/or altered before, during, and after the actual I/O operation.

Pragmatics

Since a Direct array can be used in performing Direct **I/O** operations, a Direct array is automatically **LONG.** There can be no more than 16 dimensions.

NOTE

Direct arrays are also "save" once they are used in any way. Arbitrary use of Direct arrays in lieu of normal arrays to avoid overlaying can seriously degrade overall system efficiency.

DOUBLE DECLARATION

Syntax

<double declaration> ::= *<local or own>* DOUBLE *<identifier list>*

Examples

DOUBLE DUBL
DOUBLE BIGNUMBER, GIGUNDOUS, DUBLPRECISION

Semantics

A *<double declaration>* is used to declare *<simple variable>s* which can be used as double values, that is, a 96-bit arithmetic entity (carried internally as 2 adjacent 48-bit words).

The *<local or own>* portion of the *<double declaration>* indicates whether the value of the specified *<simple variable>* is to be retained upon exit from the *<block>* in which it is declared. A *<simple variable>* is to be retained upon exit from the *<block>* in which it is declared. A *<simple variable>* declared to be **OWN** will retain its value when the program exits from the associated *<block>*, and that "old" value will be the contents of the *<simple variable>* when the associated *<block>* is re-entered.

Upon entry to a *<block>* containing *<simple variable>s*, the normal content of a non-**OWN DOUBLE** *<simple variable>* is a zero (0); i.e., two 48-bit words with all bits off. To be truly compatible with **ALGOL-60**, a programmer would explicitly zero the *<simple variable>s* with appropriate *<assignment statement>s*.

Pragmatics

After an arithmetic calculation, the resulting value is stored "as is" into the *<simple variable>*.

Appendix B contains additional information on the internal structure of a double *<simple variable>* as implemented on the **B 7000/B 6000** Information Processing System.

DUMP DECLARATION

Syntax

```
<dump declaration> ::= DUMP <dump part>
<dump part > ::= <file identifier> ( <dump list> ) <control part> |
                 <dump part> , <file identifier> ( <dump list> ) <control part>
<dump list> ::= <dump element> |
                <dump list> , <dump element>
<dump element> ::= <simple variable> |
                   <array identifier> |
                   <label identifier>
<control part> ::= <label identifier> <label counter modulus> <dump parameters>
<label counter modulus> ::= <empty> |
                            : <unsigned integer>
<dump parameters> ::= <empty> |
                      ( <label counter> <bounds part> )
<label counter> ::= <empty> |
                    <simple variable>
<bounds part> ::= <empty> |
                  ,<lower limit> |
                  ,,<upper limit> |
                  ,<lower limit> , <upper limit>
<lower limit> ::= <arithmetic expression>
<upper limit> ::= <arithmetic expression>
```

Examples

```
DUMP FYLE (A) LBL
DUMP FID (X, Y, ARAY, COWNTER) LOUP : 3
DUMP PRNTR (I, INFO, INDX) NEXT (DMPCOUNT, , DPHIGH)
DUMP LP (A, B, LBL1, ARAY) AGAIN : 5 (TALY, 20, 50)
DUMP LINEOUT (MISC, ITEM, ACCUM) MORE : 10 (DPCT, DPLOW, DPHIGH)
```

Semantics

The *<dump declaration>* allows surveillance of designated variables during execution of the user's program. The *<dump declaration>* declares which identifiers are to be placed under surveillance. Diagnostic information requested by the *<dump declaration>* is written on the file designated by the *<file identifier>* when the *<control part>* parameters are satisfied, that is:

a. If the *<label counter modulus>* is *<empty>* and *<dump parameters>* is *<empty>*, then a dump of the *<dump list>* occurs every time execution control has passed to the *<statement>* indicated by the *<label identifier>* in the *<control part>*.
b. If the *<label counter modulus>* is not *<empty>* and the *<dump parameters>* is *<empty>*, a dump of the *<dump list>* occurs whenever "n **MOD** *<label counter modulus>* = 0", where "n" is the number of times control has occurred at the label designated by *<label identifier>*.
c. If the *<label counter>* is *<empty>*, the number of times execution control has passed to the *<label identifier>* must be greater than or equal to the *<lower limit>* and less than or equal

to the *<upper limit>*, and the number of times execution control has passed to the *<label identifier>* must be evenly divisible by the *<label counter modulus>*.

d. If the *<label counter>* is not *<empty>*, the value of the designated *<simple arithmetic variable>* is used to regulate the dumping. A dump of the *<dump list>* will occur when the value of the *<label counter>* lies between the *<lower limit>* and the *<upper limit>* (inclusive), and the number of times execution control has passed to the *<label identifier>* is evenly divisible by the *<label counter modulus>*.
e. If *<label counter modulus>* is *<empty>*, 1 is assumed.
f. If *<lower limit>* is *<empty>*, zero is assumed.
g. If *<upper limit>* is *<empty>*, infinity is assumed.

Pragmatics

The diagnostic information produced depends on the form(s) of the *<dump element>s.* When a dump of the *<dump list>* occurs, the symbolic name (up to six characters) of each *<dump item>* is produced, along with the following information:

a. If the *<simple variable>* is of type **REAL, DOUBLE,** or **ALPHA,** a real value is printed. For example, **REEL** = .10000000000 or **DUBL** = 0.0 or **ALFA** = 12698307.000. If the *<simple variable>* is of type **INTEGER**, an integer value is printed. For example, **INTEGER** = 2. If the *<simple variable>* is of type **BOOLEAN**, the Boolean condition is printed. For example, **BOOL = .FALSE..**
b. A dumped *<array identifier>* of an array of type **REAL** produces the 48 bits of each array element, converted to a numeric value as if operated upon by the **R** *<editing phrase type>*. If the array is of type **BOOLEAN,** the condition of each element is shown as **.TRUE.** or **.FALSE..** If the array is of type **INTEGER**, an integer is produced for each element position.
c. A dumped *<label identifier>* shows the number of times execution control has passed the specified *<label identifier>*, For example, **L2=3.**

Restrictions

The *<array identifier>* must be that of a single-dimensioned array. Only the first six characters of any *<identifier>* are produced. *<character type>* arrays cannot be used in the *<dump declaration>*.

EVENT and EVENT ARRAY DECLARATIONS

Syntax

<event declaration> ::= **EVENT** *<event identifier list>*
<event identifier list> ::= *<event identifier>* |
<event identifier list> , *<event identifier>*
<event identifier> ::= *<identifier>*
<event array declaration> ::= **EVENT ARRAY** *<event segment list>*
<event segment list> ::= *<event segment>* |
<event segment list> , *<event segment>*
<event segment> ::= *<event array identifier list>* [*<bound pair list>*]
<event array identifier list> ::= *<event array identifier>* |
<event array identifier list> , *<event array identifier>*
<event array identifier> ::= *<identifier>*

Examples

```
EVENT FILEA
EVENT E1, E2, E3, E4
EVENT ARRAY SWAPPEE [0:5]
```

Semantics

An *<identifier>* declared to be an *<event identifier>*, or an element of an event array is usually used for purposes of synchronization. An event can be used either to indicate the completion of an activity (e.g., the completion of a Direct **I/O** read or write operation) or as an interlock between participating programs over the use of a shared resource(s).

Events can be used in synchronous manner by explicitly testing the state of the event at various programmer-defined points during execution, or they can be used in an asynchronous manner by use of the software interrupt facility.

Refer to *<event statement>* and *<interrupt declaration>*.

Pragmatics

The initial state of an event is not-happened (**RESET**) and available.

There can be no more than 16 dimensions.

FILE DECLARATION

Syntax

<file declaration> ::= *<direct specifier>* **FILE** *<file list>*
<file list> ::= *<file list part>* |
<file list> , *<file list part>*
<file list part> ::= *<file identifier>* |
<file identifier> (*<initial attribute list>*)
<file identifier> ::= *<identifier>*
<initial attribute list> ::= *<initial attribute>* |
<initial attribute list> , *<initial attribute>*

<initial attribute> ::= *<arithmetic-valued file attribute name>* = *<arithmetic file attribute value>* |
<Boolean-valued file attribute name> |
<Boolean-valued file attribute name> = *<Boolean expression>* |
<pointer-valued file attribute name> = *<pointer expression>* |
<pointer-valued file attribute name> = *<string>*
<translate-table-valued file attribute name>
<arithmetic file attribute value> ::= *<arithmetic expression>* |
<mnemonic file attribute value>
<arithmetic-valued file attribute name> ::= **AREACLASS** | **AREAS** | **AREASIZE** |
ASSIGNTIME | **ATTVALUE** | **ATTYPE** |
BLOCK | **BLOCKSIZE** | **BUFFERS** |
CARRIAGECONTROL | **CENSUS** | **COPIES** |
CURRENTBLOCK | **CYCLE** | **DATE** | **DENSITY** |
DIRECTION | **DISPOSITION** | **ERRORTYPE** |
EXTMODE | **FAMILYSIZE** | **FILEKIND** |
FILETYPE | **INTMODE** | **KIND** | **LABELTYPE** |
LASTRECORD | **LASTSTATION** | **LINENUM** |
MAXRECSIZE | **MINRECSIZE** | **MYUSE** | **PAGE** |
PAGESIZE | **PARITY** | **POPULATION** |
PROTECTION | **RECEPTIONS** | **RECORD** |
RECORDINERROR | **RECORDKEY** | **REEL** |
ROWADDRESS | **ROWSINUSE** | **SAVEFACTOR** |
SECURITYTYPE | **SECURITYUSE** | **SERIALNO** |
SIZEMODE | **SIZEOFFSET** | **SIZE2** | **SPEED** | **STATE** |
TAPEREELRECORD | **TRANSLATE** | **TRANSMISSIONO** |
TRANSMISSIONS | **UNITNO** | **UNITS** | **UNITSLEFT** |
USEDATE | **VERSION** | **WIDTH**

<mnemonic file attribute value> ::= *<density mnemonic>* |
<errortype mnemonic> |
<extmode mnemonic> |
<filekind mnemonic> |
<intmode mnemonic> |
<kind mnemonic> |
<labeltype mnemonic> |
<myuse mnemonic> |
<parity mnemonic> |

<protection mnemonic> |
<securitytype mnemonic> |
<securityuse mnemonic> |
<sizemode mnemonic> |
<speed mnemonic> |
<state mnemonic> |
<translate mnemonic> |
<units mnemonic>

<density mnemonic> ::= **HIGH | MEDIUM | LOW | SUPER**

<errortype mnemonic> ::= **NOERROR | SUNOTREADY | READPARITYERROR | READCHECKFAILURE**

<extmode mnemonic> ::= **SINGLE | HEX | BCL | EBCDIC | ASCII | BINARY**

<filekind mnemonic> ::= **ALGOLCODE | ALGOLSYMBOL | BACKUPDISK | BACKUPPRINTER | BACKUPPUNCH | BASICCODE | BASICSYMBOL | BINDERSYMBOL | BOUNDCODE | CDATA | COBOLCODE | COBOLSYMBOL | CODEFILE | COMPILERCODEFILE | CONTROLDECK | CSEQDATA | DATA | DCALGOLCODE | DCALGOLSYMBOL | DIRECTORY | ESPOLCODE | ESPOLSYMBOL | FORTRANCODE | FORTRANSYMBOL | GUARDFILE | INTRINSICFILE | JOBCODE | JOBDESCFILE | JOVIALCODE | JOVIALSYMBOL | LIBRARYCODE | MCPCODEFILE | PLICODE | PLISYMBOL | RECONSTRUCTIONFILE | SEQDATA | SYSTEMDIRECTORY | SYSTEMDIRFILE | VERSIONDIRECTORY | XALGOLCODE | XALGOLSYMBOL | XDISKFILE | XFORTRANCODE | XFORTRANSYMBOL**

<intmode mnemonic> ::= **ASCII | BCL | EBCDIC | HEX | SINGLE**

<kind mnemonic> ::= **CP | DC | DISK | DISKPACK | DISPLAY | PACK | PAPER | PAPERPUNCH | PAPERREADER | PETAPE | PRINTER | PTP | PTR | PUNCH | READER | REMOTE | SPO | TAPE | TAPE7 | TAPE9**

<labeltype mnemonic> ::= **STANDARD | OMITTED | OMITTEDEOF**

<myuse mnemonic> ::= **CLOSED | IN | OUT | IO**

<parity mnemonic> ::= **STANDARD | NONSTANDARD**

<protection mnemonic> ::= **TEMPORARY | SAVE | PROTECTED**

<securitytype mnemonic> ::= **PRIVATE | CLASSA | CLASSB**

<securityuse mnemonic> ::= **SECURED | IN | OUT | IO**

<sizemode mnemonic> ::= **SINGLE | HEX | BCL | EBCDIC | ASCII**

<speed mnemonic> ::= **FAST | MEDIUMFAST | MEDIUMSLOW | SLOW**

<state mnemonic> ::= **ATEND | |**
BREAKHERE |
DATAERROR |
LOCKEDOUT |
NEWUSER |
NOINPUT |

NORMAL |
PARITYERROR |
TIMEOUT
<*translate mnemonic*> ::= **DEFAULTTRANS** |
FORCESOFT |
FULLTRANS |
NOSOFT |
NOTRANS |
SOFTONLY
<*units mnemonic*> ::= **CHARACTERS** | **WORDS**
<*Boolean-valued file attribute name*> ::= **ATTERR** |
CYLINDERMODE | **DUPLICATED** | **ENABLEINPUT** | **EOF** |
FLEXIBLE | **INTERCHANGE** | **NULINPUT** | **OPEN** |
OPTIONAL | **PRESENT** | **READCHECK** | **RESIDENT** |
SCREEN | **SINGLEPACK** | **TRANSLATING** | **UPDATED**
<*pointer-valued file attribute name*> ::= **FAMILY** | **FORMMESSAGE** |
INTNAME | **PACKNAME** | **TITLE**
<*translate-table-valued file attribute name*> ::= **INPUTTABLE** | **OUTPUTTABLE**

Examples

```
FILE FYLE
FILE TAPE (KIND=DISK,FILETYPE=8, BUFFERS=2,
     INTMODE=EBCDIC)

FILE OFNI (KIND=DISK,BUFFERS=3,AREASIZE=30,
     MAXRECSIZE=246, BLOCKSIZE=2560,
     AREAS=100,
     TITLE="INFO.")
```

Semantics

A <*file declaration*> associates a <*file identifier*> with a file. The attributes for that particular file may or may not be specified in the <*file declaration*>. The attributes not specified in the <*file declaration*> can be assigned by an appropriate <*assignment statement*> or through the use of Work Flow Language statements at either compile-time or execution-time.

Any pointer <*initial attribute*> can be set equal to a character string constant as well as a <*pointer expression*>.

A <*Boolean-valued file attribute name*> appearing without the "= <*logical value*>" part implies "= **TRUE**."

Pragmatics

There are two methods of performing **I/O** operations on the **B 7000/B 6000** Information Processing System. The first method is the simplest and is referred to as "normal **I/O**" or as "regular **I/O**". (Refer to <**I/O** *statement*>.)

Normal **I/O** is indicated when the *<direct specifier>* is *<empty>*. Normal **I/O** includes many automatic facilities provided by the MCP, such as:

a. Buffering – the automatic overlap of program processing and **I/O** traffic from/to the peripheral units.
b. Blocking – more than one logical record per physical block.
c. Translation – as needed between the character set of the unit and that required by the program.

Direct **I/O** is the indicated method when **DIRECT** is specified. The functions of buffering, blocking, and translation (as described above) become the responsibility of the programmer. Furthermore, a **DIRECT ARRAY** is required in order to **READ** from and/or **WRITE** to the specified file. Refer to the *<read statement>* and *<write statement>*.

Both Non-Direct files and Direct files have numerous file attributes which can be interrogated and/or altered. Direct **I/O** files have a number of additional attributes which are pertinent to Direct **I/O** only. (Refer to the **B 7000/B 6000** Input/Output Subsystem Reference Manual, form 5000185.)

FORMAT DECLARATION

Syntax

```
<format declaration> ::= FORMAT <in-out part> <format part list>
<in-out part> ::= <empty> |
                  IN |
                  OUT
<format part list> ::= <format part> |
                       <format part list> , <format part>
<format part> ::= <format identifier> ( <editing specifications> ) |
                  <format identifier> 〈 <editing specifications> 〉
<format identifier> ::= <identifier>
<editing specifications> ::= <editing segment> |
                             <editing specifications>/ |
                             / <editing specifications> |
                             <editing specifications> / <editing segment> |
                             <editing specifications> , <editing segment>
<editing segment> ::= <editing phrase> |
                      <repeat part> ( <editing specifications> ) |
                      <editing segment> , <repeat part> ( <editing specifications> )

<editing phrase> ::= <repeat part> <editing phrase type> <field width part>
<repeat part> ::= <empty> |
                  <unsigned integer> |
                  *
<editing phrase type> ::= <simple string> |
                          A | C | D | E | F | G | H | I | J | K | L | O | P | R |
                          S | T | U | V | X | Z | $
<field width part> ::= <empty> |
                       <field width> <decimal places>
<field width> ::= <unsigned integer> | *
<decimal places> ::= <empty> |
                     . <unsigned integer> |
                     . *
```

Examples

```
FORMAT HDG ("THIS REPORT SHOULD BE MAILED TO ROOM W-252")
FORMAT IN EDIT (X4, 2I6, 5E9.2, 3F5.1, X4)
FORMAT IN F1 (A6,5(X3,2E10.2,2F6.1),317),F2(A6,G,A6)
FORMAT OUT FORM1 (X56, "HEADING",X57),FORM2 (X10,4A6/X7,
    5A6/X2,5A6)
FORMAT FMT1 (*I*)
FORMAT FMT2 (*V*.*)
```

Semantics

A <format declaration> associates each of its <format identifier>s with an <editing specifications>. <define identifier>s, <remark>s, and <formal symbol>s cannot be used in formats.

A format can be referenced in a *<read statement>*, *<write statement>*, or a *<switch format declaration>*. In general, a *<list>* would also be referenced in those same statements, and the joint purpose is to indicate a series of data items (specified by the *<list>*) along with the formatting action (specified by the *<format identifier>*) to be performed on each of the data items.

<in-out part>

The *<in-out part>* has effect only upon the treatment of *<simple string>s* used with a format. Under certain circumstances a *<simple string>* (appearing as an *<editing phrase type>*) is read-only. Any attempt to store into read-only entity results in a program execution error.

If the *<in-out part>* of a *<format declaration>* is **OUT** or *<empty>*, there is a run-time error if an attempt is made to replace any *<simple string>* in the format via a *<read statement>*. If the *<in-out part>* is **IN**, *<simple string>s* within formats are not read-only and can be replaced. However, once a *<simple string>* has been replaced, the format containing it is altered from its original definition in the *<format declaration>*. When reading data into a format element to replace a *<simple string>*, no more characters can be transferred than appear in the *<simple string>*.

SLASH

Two fields in a format item list are separated by a comma, a slash, or a series of slashes. A slash is used to indicate the end of a record. On input, any remaining characters in the current record are ignored when a slash is encountered in the specification list. On output, the construction of the current record is terminated and any subsequent output is placed in the next output record(s). Multiple slashes may be used to skip several records of input or generate several blank records on output. The final right parenthesis of a format also acts to indicate the end of the current record.

Carriage control occurs each time a slash appears in the format. With the *<core-to-core file part>*, a slash in the format is ignored.

Example

```
<I> COMPILE FMT/TEST ALGOL; EBCDIC
    BEGIN
    FILE READER (KIND=READER) ,
         LINE     (KIND=PRINTER) ;
    REAL A, B;
    FORMAT FMT(I2,/,I2);
    READ (READER, FMT, A,B);
    WRITE(LINE, FMT, A,B);
    WRITE(LINE [SKIP 1],FMT, A,B);
    END.
<I> DATA
 1234
 5678
<I>END
```

Produces the following output:
12
56
12
•
•
• <to channel 1>
56

NOTE

For ease of explanation, lower case letters are used to refer to the parts of an *<editing phase>*:

r = *<repeat part>*
w = *<field width>*
d = *<decimal places>*

Asterisks

If an asterisk (*) appears in a format specification list in place of the r, w, or d parts, then the I/O list will be accessed once and the value of the I/O list element obtained will be used to replace the *. A new I/O list element is required each time an * is encountered in the specification list.

<repeat part>

Format specifications and format list portions enclosed in parentheses may optionally be immediately preceded by an unsigned nonzero integer constant. This constant indicates the number of times that portion of the specification list is to be interpreted. If no such repeat count is indicated, a repeat count of 1 is assumed.

If the outer right parenthesis of the format specification list is encountered before the I/O list is exhausted, control reverts to the repeat count (if present) of the repeat specification group terminated by the last preceding right parenthesis. If no other right parenthesis exists in the specification list, then control reverts to the first left parenthesis of the specification list.

The following are proper examples of the use of repeat counts. In each case, the repeat count is 3.

```
3F10.4
3(A6/)
3(3A6,3(/I12)/)
```

If the *<repeat part>* is *<empty>*, a value of 1 is assumed.

If the *<repeat part>* is an *, the number of repetitions is determined by the value of the corresponding *<list element>* as follows:

a. If the value is greater than 0, then repeat the number of times represented by the value.

b. If the value is equal to 0, then repeat indefinitely.
c. If the value is less than 0, then skip to the corresponding right parenthesis.

Example

```
<I> COMPILE VAR/REPEAT ALGOL; EBCDIC
     BEGIN
     FILE LINE(KIND=PRINTER);
     REAL A,B,C;
     FORMAT FMT(*(A2,X1),*I2);
     A:=1; B:=2; C:=3;
     WRITE(LINE,FMT,2,"AB","CD",3,A,B,C);
     WRITE(LINE,FMT,-3,1,A);
     WRITE(LINE,FMT,0,"AB","CD","EF");
END.
<I>END
Produces the following output:
ABbCDbb1b2b3
b1
ABbCDbEFb
```

<width part>

When an asterisk used for the field width of a format phrase is given a zero or negative value at run-time, no editing action occurs for that phrase; however, the next list element is skipped as if it had been edited by the inactive editing phrase. (If a zero or negative field width occurs (at run-time) for a phrase with a repeat part, enough list elements are skipped to satisfy the repeat count.)

Example

```
<I> COMPILE VAR/WIDTH ALGOL; EBCDIC
     BEGIN
     FILE LINE(KIND=PRINTER);
     REAL A;
     FORMAT FMT(I*,A*);
     A:=12;
     WRITE(LINE,FMT,3,A);
     WRITE(LINE,FMT,0,A,2,"AB");
     END.
<I>END
Produces the following output:
b12
AB
```

Editing Phrase Actions

The actions of the various *<editing phrase type>s* are explained in the following information, arranged in alphabetical order according to the *<editing phrase type>* letter.

<*simple string*> Format

The presence of a <*simple string*> in a format indicates that the characters enclosed by the quote marks (") are to be used as the data. The occurrence of a <*simple string*> does not require a corresponding <*list element*> when the format is used.

BCL strings (those with string codes of 6 or 3) are encoded as **BCL** characters, not **EBCDIC** characters.

To enable more efficient handling of string codes in formats, 1-bit, 2-bit and 7-bit strings are not allowed. If no string code appears with a quoted string, the default character size (6-bit if the **BCL** compiler option is set; 8-bit otherwise) will be used. The length of the 3-bit and 4-bit strings must be a multiple of 2 to facilitate packing into 6-bit or 8-bit characters, respectively. Only the first digit of the string code is ever used when encoding formats, since the extra information available in string code is meaningless in the case of formats.

Example

```
      .
      .
   WRITE(LINE,<4"C1C2", 8"ABC">);
$SET BCL
   WRITE(LINE,<3"646566", 6"HIJ">);
      .
      .
```

Will produce the following output:
ABABC
DEFHIJ

A Format

The alphanumeric format specification Aw causes data to be transferred to or from internal storage as **EBCDIC** (8-bit) or **BCL** (6-bit) characters.

NOTE

Prior to II.7, the **INTMODE** of the file determined the character size applied to list elements (except pointers). On II.7, the default character size (6-bit if **$SET BCL** appears, 8-bit otherwise) applies to list elements (other than pointers). This gives the added flexibility of writing **BCL** (6-bit) data to an **EBCDIC** (8-bit) file (and vice versa) and similarly for input, with translation occurring where necessary to preserve character data.

Example

```
BEGIN
FILE F(KIND=PRINTER,INTMODE=EBCDIC);
WRITE(F, <A3>,8"ABC");
$SET BCL
WRITE(F, <A3>, 6"ABC");
END.
```

Output prior to II.7:

```
ABC
???     (where ? represents a nongraphic EBCDIC character)
```

OUTPUT on II.7:

```
ABC
ABC
```

Pointers

On input, w characters are transferred from the input record to the pointer-designated location. On output, w characters are transferred from the pointer-designated location to the output record. The *<character size>* used is that of the pointer.

NOTE

For purposes of explanation of A and C formats, the variable Q will be used, where the value of Q is derived from the following table:

		(default character size) BCL	EBCDIC
(precision)	Single	8	6
	Double	16	12

[if the list element is *<pointer expression>* **FOR** *<arithmetic expression>*, use the *<arithmetic expression>* as the value of Q.]

Input

On input, the A-format specification causes the character string of width w in the external field to be assigned to the corresponding simple variable or array element in the I/O list. Legal *<list element>s* are of type **ALPHA, INTEGER, BOOLEAN, DOUBLE, REAL,** or **POINTER**.

If w is greater than or equal to Q, the right-most Q characters of the input field are transferred to the *<list element>*. If w is less than Q, w characters of the input field are transferred to the *<list element>*, right-justified. The unused high-order bits of the data word are set to zero.

Input Examples

DEFAULT CHARACTER SIZE	EXTERNAL STRING	SPECIFICATION	INTERNAL VALUE
8	ABCDEFGHIJKL	A9	8"DEFGHI"
6	ABCDEFGHIJKL	A9	6"BCDEFGHI"
8	AbCbEbGbIbK	A4	4"0000"8"AbCb"
6	ABCDEFGHIJKL	A4	6"0000ABCD"
(either)	ABCDEFGHIJKL	A12	ABCDEFGHIJKL (pointer as *<list element>*)
8	ABCDEFGHIJKL	A12	4"0000"8"ABCDEFGHIJKL" (8-bit pointer FOR 14)
6	ABCDEFGHIJKL	A12	6"JKL" (6-bit pointer FOR 3)

NOTE

If the corresponding *<list element>* is an **INTEGER** variable, the w characters of the input field are stored into this *<list element>* without integerization being performed. If w is greater than 4, the **INTEGER** *<list element>* can receive a noninteger value. (Refer to Word Formats in appendix B.)

Output

On output, the A *<editing phrase>* causes the characters contained in the appropriate variable in the *<list element>* to be converted into an external string of length w.

If w is greater than or equal to Q, the Q characters of the *<list element>* are placed right-justified in the field, preceded by w minus Q blanks.

If w is less than Q the right-most w characters of the *<list element>* are written into the output field. If the output character size is 8-bit and one of the character fields in the word contains a bit pattern that does not correspond to an **EBCDIC** graphic,? (denoting an invalid character) would be printed in that position.

Output Examples

DEFAULT CHARACTER SIZE	INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
8	8"DEFGHI"	A9	bbbDEFGHI
6	6"BCDEFGHI"	A9	bBCDEFGHI
8	4"0000000000"8"A"	A4	???A
6	6"0000ABCD"	A4	ABCD
8	8"ABCDEFG" (8-bit pointer FOR 7)	A11	bbbbABCDEFG
6	6"ABCDEFG" (6-bit pointer FOR 7)	A4	DEFG

C Format

The Cw format specification has the same effect as the Aw format specification except that characters are placed into and taken from the leftmost portion of a word (or list element).

Input Examples

DEFAULT CHARACTER SIZE	EXTERNAL STRING	SPECIFICATION	INTERNAL VALUE
8	ABCDEFGHIJKL	C9	8"DEFGHI"
6	ABCDEFGHIJKL	C9	6"BCDEFGHI"
8	ABCD	C4	8"ABCD"4"0000"
6	ABCDEFGHIJKL	C4	6"ABCD0000"
8	ABCDEFGHIJKL	C12	8"ABCDEFGHIJKL"4"0000" (8-bit pointer FOR 14)
6	ABCDEFGHIJKL	C12	6"JKL" (6-bit pointer FOR 3)

Output Examples

DEFAULT CHARACTER SIZE	INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
8	8"DEFGHI"	C9	bbbDEFGHI
6	6"BCDEFGHI"	C9	bBCDEFGHI
8	8"ABCD"4"0000"	C5	ABCD?
6	6"ABCD0000"	C4	ABCD
8	8"ABCDEFG" (8-bit pointer FOR 7)	C11	bbbbABCDEFG
6	6"ABCDEFG" (6-bit pointer FOR 7)	C4	ABCD

D,E Formats

The format specifications Dw.d and Ew.d cause data appearing in an external character string as a numeric constant to be associated with an internal storage location for purposes of input or output.

Correct action will occur for list elements of type **ALPHA, INTEGER, REAL, DOUBLE** or **BOOLEAN.**

Input

[In the following discussion and examples for input, the letter "D" may be substituted wherever "E" is used.]

On input, the Ew.d specification causes the value of the numeric constant written with or without exponential notation in a string of w input characters to be assigned to the corresponding I/O list element.

The Ew.d specification allows the input constant to contain as many decimal places as desired by use of the decimal place count, d. If no decimal point appears in the input string, a decimal point is implied as specified by d. Thus, the input string 100E0 when read using the specification E5.2 would be interpreted as the numeric constant 1.E+0 with two implied decimal places in the input string. A decimal point is assumed d places from either the right edge of the input field or from the E denoting the exponent, if there is one.

The field width, w, must be greater than or equal to the specified number of decimal places, d. A blank is interpreted as a zero.

Examples

EXTERNAL STRING	SPECIFICATION	INTERNAL VALUE
bbbbbb25046	E11.4	+2.5046
bbbbb25.046	E11.4	+25.046
-bb25046E-3	E11.4	-0.0025406
bb250.46E-3	E11.4	+0.25046
b-b25.04678	E11.4	-25.04678

Output

On output, the Dw.d and Ew.d specifications cause the value of the corresponding item in the I/O list to be written as an output character string of length w, representing a numeric constant expressed in exponential notation. The exponent is adjusted so that the decimal point is positioned as specified by the decimal place count, d.

The specified width of the output field, w, must be greater than or equal to the number of specified decimal places, d, plus 7. This provides for a 4-character exponent part, a decimal point, a digit preceding the decimal point, and a sign. If this rule is violated, the field will be filled with asterisks.

The Dw.d specification is essentially equivalent to the Ew.d specification except for the presence of a D rather than an E in the exponent part of the output string.

Furthermore, the number of characters necessary to represent the D exponent part depends upon the value of the exponent. The following types of exponent parts may appear:

(4-character)	D±XX	where	01⩽XX⩽99
(4-character)	±XXX	where	100⩽XXX⩽999
(7-character)	D±XXXXX	where	01000⩽XXXXX⩽99999

Output Examples

INTERNAL VALUE	SPECIFICATIONS	EXTERNAL STRING
+36.7929	E13.5	bb3.67929Eb01
-36.7929	E12.5	-3.67929Eb01
-36.7929	E11.5	3.67929Eb01
+36.7929	E10.5	**********
1.234@@-73	D14.5	bbb1.23400D-73
-789@@1234	D15.3	bb-7.890D+01236
6.54@@321	D9.2	b6.54+321

F Format

The real format specification Fw.d causes data appearing in an external character string as a real constant to be associated with an internal storage location for purposes of input or output. Correct action will occur for list elements of type **ALPHA, INTEGER, REAL, DOUBLE**, or **BOOLEAN.**

On input, the Fw.d specification causes the value of the real constant written with or without exponential notation in a string of w input characters to be assigned to the corresponding I/O list element.

The decimal point may be positioned as indicated in the input string or located as desired via the decimal place count, d. If no decimal point appears in the input string, a decimal point is implied as specified by d. A decimal point is assumed d places from the right edge of the input field. Thus, the input string 1234 when read using the specification F4.2 would be interpreted as the real constant 12.34 with two implied decimal places in the input string.

The field width, w, must be greater than or equal to the specified number of decimal places, d, and must include the decimal point and exponent field when either or both are present. A blank is interpreted as a zero.

Examples

EXTERNAL STRING	SPECIFICATION	INTERNAL VALUE
36725931	F8.4	+3672.5931
3.672593	F8.4	3.672593
-367259.	F8.4	-367259
-3672.E2	F8.4	-367200
367259E2	F8.4	+3672.59
3.672E-1	F8.4	+.3672
367259	F6.6	+0.367259
b-b3456	F7.2	-34.56

Output

On output, the Fw.d specification causes the value of the corresponding item in the I/O list to be written as an output character string of length w, representing a real constant expressed without using exponential notation. The decimal point is adjusted such that d digits follow the decimal point.

The constant is right-justified over blanks within the field, and the specified width of the output field, w, must be greater than or equal to the number of specified decimal places, d, plus 1. The possible presence of a minus sign for a negative datum must be taken into consideration when specifying the field width.

The internal value is rounded to satisfy the decimal point specification, and the field will contain asterisks if the value to be output has an integer part too large for the allotted field.

Examples

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
+36.7929	F7.3	b36.793
+36.7934	F9.3	bbb36.793
-0.0316	F6.3	-0.032
0.0	F6.4	0.0000
0.0	F6.2	bb0.00
+579.645	F6.2	579.65
+579.645	F4.2	****
-579.645	F6.2	******

G Format

The *<field width part>* must be *<empty>*. No *<list element>* corresponds to this editing letter.

BCL Files

On input, eight 6-bit characters from the input record are skipped. On output, eight **BCL** zeroes are written.

EBCDIC Files

On input, six 8-bit characters from the input record are skipped. On output, six **EBCDIC** zeroes are written.

H, K Formats

NOTE

[For purposes of explanation of H and K formats, the variable Q will be used, where the value of Q is derived from the following table:

		(format phrase)	
		H	K
(precision)	single	12	16
	double	24	32

Also, the term Characters will refer to hexadecimal characters for H format, and octal characters for K format.]

The Hw and Kw format specifications cause an external string of Characters in a field of width w to be interpreted as a hexadecimal (H) or octal (K) value and associated with the corresponding list element for purposes of input data transfer. Conversely, an internal value is converted to Characters and associated with a corresponding list element for purposes of output data transfer. Legal list elements are of type **ALPHA, REAL, INTEGER, DOUBLE** and **BOOLEAN.**

Input

On input, the value represented by the Characters in the input field is assigned to the corresponding *<list element>* variable. Leading, trailing and embedded blanks are interpreted as zeroes. A minus (-) sign causes bit 46 of the storage word (or the first word of a double) allocated to the variable to be complemented.

If the input data is less than or equal to Q Characters long, it is stored right-justified in the storage location (both words of a double are included). Unused high-order bits are set to zero. If w is greater than Q, the leftmost w minus Q Characters must be blank, zero or minus; otherwise a data error will occur.

Input Examples

EXTERNAL STRING	SPECIFICATION	INTERNAL VALUE
6F	H2	4"00000000006F"
1FFFFFFFFFFF	H12	4"1FFFFFFFFFFF"
-16	H3	4"400000000016"
1234b568	H8	4"000012340568"
FFCb	H4	4"00000000FFC0"
00C1C2C3C4C5C6	H14	4"C1C2C3C4C5C6"
-ABCD	H5	4"40000000000000000000ABCD" (double)
123456789ABCDEF	H15	4"000000000123456789ABCDEF" (double)
16	K2	3"0000000000000016"
1777777777777777	K16	3"1777777777777777"
-16	K3	3"2000000000000016"
1234b56	K7	3"0000000001234056"
77b	K3	3"0000000000000770"
-567	K4	3"20000000000000000000000000000567" (double)
1234567654321234567	K19	3"00000000000001234567654321234567" (double)

NOTE

If the input string contains a non-Character, an error occurs, and the "data error" *<action label>* of the *<read statement>* is invoked (if specified).

Output

On output, the value of the *<list element>* is printed as a string of Characters right-justified over blanks in a field of width w. If w is less than Q, the contents of the rightmost w*4 bits (H) or w*3 bits (K) of the storage word (consider a double-precision variable as effectively a 96-bit word) are printed as a string of w Characters. If w is greater than Q, the Q Characters of the *<list element>* are placed right-justified in the output field, preceded by w minus Q leading blanks. Such output never contains a printed sign.

Output Examples

INTERNAL VALUE	SPECIFICATION	EXTERNAL VALUE
4"0000E5551010"	H5	51010
4"0000E5551010"	H12	0000E5551010
4"0000E5551010"	H16	bbbb0000E5551010
8"123456"	H12	F1F2F3F4F5F6
4"00000000000000012345678" (double)	H4	5678
8"123456789bbb" (double)	H24	F1F2F3F4F5F6F7F8F9404040
3"0005677701234445"	K5	34445
3"0005677701234445"	K16	0005677701234445
3"0005677701234445"	K18	bb0005677701234445
3"00000000000000000000001234567" (double)	K4	4567

I Format

The integer format specification Iw causes an external character string of width w to be associated with the corresponding list element for purposes of data transfer. Legal list elements are of type **ALPHA, REAL, INTEGER, DOUBLE**, or **BOOLEAN.**

Input

On input, the Iw specification causes the value of the integer constant in the input field to be assigned to the corresponding list element. Any legal **ALGOL** integer constant is allowed in the field. Blank characters are interpreted as zeroes. The magnitude of the value which may be input depends upon the type of the list element.

Input Examples

EXTERNAL STRING	SPECIFICATION	INTERNAL VALUE
567	I3	+567
bb-329	I6	-329
-bbbb27	I7	-27
27bbb	I5	+27000
b-bb234	I7	-234

Output

On output, the Iw specification causes the value of the corresponding list element to be printed as an integer constant in a field of width w. The constant is right-justified over a field of blanks, and the plus sign is not printed for non-negative quantities.

If the value of the list element requires a field larger than w, then w asterisks will be printed.

Floating-point values are rounded to an integer value before printing.

Output Examples

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
+23	I4	bb23
-79	I4	b-79
+67486	I5	67486
-67486	I5	*****
+978	I1	*
0	I3	bb0
+3.6	I2	b4

J Format

The integer format specification Jw causes an external character string of at most w characters to be associated with the corresponding list element for purposes of data transfer. Legal list elements are of type **ALPHA, REAL, INTEGER, DOUBLE,** or **BOOLEAN.**

Input

On input, the Jw specification functions identically to the Iw specification.

Output

On output, the Jw specification causes the value of the corresponding list element to be printed as an integer constant in the minimum field necessary to contain the value without exceeding w. The plus sign is not printed for non-negative quantities.

If the value to be printed requires more than w characters, w asterisks will be printed.

Floating-point values are rounded to an integer value before printing.

Output Examples

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
+23	J5	23
-23	J5	-23
+233	J3	233
-233	J3	***
0	J3	0

K Format

[K format is discussed in conjunction with H format.]

L Format

The logical format specification Lw causes the logical value indicated by the contents of a character string of width w to be associated with the corresponding list element for purposes of data transfer. Legal list elements are of type **ALPHA, REAL, INTEGER, DOUBLE**, or **BOOLEAN.**

Input

On input, the Lw specification causes the corresponding list element to be assigned the value TRUE (1) or FALSE (0), depending on the contents of the field of width w. If the left-most non-blank character is a T, the variable is assigned the value TRUE; otherwise, the value FALSE is assigned. An all-blank field yields the value FALSE. If the list element is a double, the first word is assigned the logical value and the second word is set to zero.

Input Examples

EXTERNAL STRING	SPECIFICATION	INTERNAL VALUE
T	L1	TRUE(4"000000000001")
bbF	L3	FALSE(4"000000000000")
bbbTRU	L6	TRUE(4"000000000001")
b	L1	FALSE(4"000000000000")
T	L1	TRUE(4"000000000001000000000000") (double)

Output

The list element may be a variable or an <*expression*>. If bit 0 of the corresponding list element (only the first word of a double is considered) is **ON** or **OFF**, the logical value of the item is **TRUE** or **FALSE**, respectively.

Output Examples

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
0	L6	bFALSE
1	L5	bTRUE
2	L4	FALS
3	L3	TRU
4	L2	FA

O Format

NOTE

[For purposes of explanation of the O format, the variable Q will be used, where the value of Q is derived from the following table:

	(precision) single	double	(pointers) 4-bit	6-bit	8-bit
BCL	8	16	12	8	6
EBCDIC	6	12	12	8	6

(default character size)

For pointers, if Q (from the table) is greater than the length (in characters) of the string pointed to, the value of Q is the string length.]

On input, Q characters are transferred, unedited, from the input record to the list element. On output, Q characters are transferred, unedited, to the output record from the list element. The *<field width part>* must be *<empty>*. Legal list elements are of type **ALPHA, REAL, INTEGER, DOUBLE, BOOLEAN**, or **POINTER.**

P,$ Formats

Format modifiers may be placed immediately to the left of a format specification used to edit a data item for output. If a repeat count is used, it should be to the left of any modifiers used. More than one modifier may be used with a format specification. A modifier may not be used on input.

For example, 2PR10.3 and 8P$F20.6 are valid, but $2F5.1 is not.

P Format Modifier

On output, this phrase may be used in conjunction with a numeric editing phrase to cause commas to be inserted between digit triples to the left of the decimal point. (This phrase is not allowed on input.)

$ Format Modifier

On output, this phrase may be used in conjunction with a numeric editing phrase to place a dollar sign immediately to the left of an edited item. (This phrase is not allowed on input.)

Examples:

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
17.347	$F10.2	bbbb$17.35
-1234567	PI10	-1,234,567
-1234567	P$Z15.2	bbbb$-1,234,567
1234567.11111	PF15.5	1,234,567.11111
1234567.1234	$PR15.5	bbb$1.23457E+06
1234567.1234	$PR15.0	bbbb$1,234,567.

R Format

The Rw.d format specification is a generalized numeric editing phrase which can be associated with an S format scale factor. Correct action will occur for list elements of type **ALPHA, REAL, INTEGER, DOUBLE** or **BOOLEAN**.

Input

On input, the contents of the input field are transferred to the list element in accordance with the D, E or F formats (subject to the effects of an S format scale factor). A "D", an "E" or an "@" can be used to indicate the beginning of the exponent field. A number with an implied exponent indicator is treated as if the exponent indicator is actually present. For example, 1.0–3 would be 1.0@–3. Blank characters are interpreted as zeroes.

Output

On output, the value of the *<list element>* is placed in the field described by the field width. The number used as the decimal exponent in the following algorithm is the exponent number of the normalized value of the *<list element>*, using scientific notation. For example, 376.42 normalized is 3.7642E2, where the 2 following the E is the decimal exponent. D format specification, E format specification, or F format specification editing is used according to the following test:

If **ABS** (*<list element>*) ≥ 1 and
w ≥ (decimal exponent+1) +1 + d + **SIGNBIT**

or **ABS** (*<list element>*) < 1 and
w ≥ d + 1 + **SIGNBIT** and
(d ≥ –(decimal exponent) or
w < d + 1 + 5 + **SIGNBIT**)

then F *<editing phrase>* editing, else

If **ABS** (decimal exponent) ≤ 99 and
w ≥ d + 6 + **SIGNBIT**,

then E *<editing phrase>* editing, else

If w > d + 9 + **SIGNBIT**,

then D <*editing phrase*> editing, else

Fill w character positions with asterisks, because w is too small.

EXTERNAL INPUT STRING	LIST ELEMENT TYPE	SPECIFICATION	EXTERNAL OUTPUT STRING
-.333333bb	REAL	R10.4	bbb-0.3333
-.333333bb	DOUBLE	R10.4	bbb-0.3333
-.333333bb	INTEGER	R10.4	bbbb0.0000
3333.333E2	DOUBLE	R10.4	3.3333D+05
3333.333E2	INTEGER	R10.4	3.3333E+05
-.333bbbbb	REAL	R10.9	**********
-.333bbbbb	INTEGER	R10.9	.000000000
333.333E2b	DOUBLE	R10.4	3.3333D+22
bbbbbbbbbbbbb1.23D12	REAL	R20.4	bb1230000000000.0000
bbbbbbbbbb1.23D12345	DOUBLE	R20.4	bbbbbbb1.2300D+12345
bbbb4.3@68	REAL	R10.4	4.3000E+68

S Format

Input

On input, the values associated with the subsequent R <*editing phrase*> are divided by the "power of 10" designated by the <*integer*> in S <*integer*>.

Output

The values associated with the subsequent R <*editing phrase*> are multiplied by the "powers of 10" designated by the <*integer*> in S <*integer*>. More than one S <*integer*> phrase can appear in a format, each phrase taking precedence over the preceding one. For example, the execution of the following program excerpt:

```
•
•
•
READ(KARD, <R10.2>, A);
•
•
WRITE(LINE, <S3,R10.2>, A);
•
•
•
```

with input data of 10.00 and .54 yields
bb10000.00 and bbb540.00 as input.

T Format

The buffer point is moved to the wth character position in the record. The *<field width>*, w, must be greater than zero (0), that is, T1 moves the buffer pointer to the first character position in the record. No *<list element>* corresponds to this editing letter.

Example:

```
<I>COMPILE T/FORMAT ALGOL; EBCDIC
    BEGIN
        FILE LINE(KIND=PRINTER), KARD(KIND=READER);
        REAL A;
        READ(KARD, <T7, A6>, A);
        WRITE(LINE, <A6, T12, A6>, A, A);
        WRITE(LINE, <X6, "123", T1, A6>, A);
  END.
<I>DATA
  ABCDEFGHIJKLMN
<I>END
```

produces the following output:

```
GHIJKLbbbbbGHIJKL
GHIJKL123
```

U Format

The U editing specification is a flexible editing phrase which allows a great deal of freedom in the transfer of formatted data. Legal list elements are of type **ALPHA, REAL, INTEGER, DOUBLE** or **BOOLEAN.**

Input

U format has yet to be implemented for input.

Output

On output, the U editing specification causes the data item to be output in a form best suited for the item. **REAL, INTEGER,** and **DOUBLE** items are output in a format that combines readability with maximum numerical significance. **BOOLEAN** items are output as "T" or "F" and occupy one character position in the record. Character strings are treated as real. If the number of characters required to edit the item is greater than the number left in the current record, the record is output and the item placed in the next record.

The form Uw is similar to U, with the added restriction that the edited term may not exceed w characters. If the data item cannot be edited into a field of w characters, a field of w asterisks is output.

The form Uw.d is similar to Uw, with the added restriction that the total field width occupied by the edited item may not be less than d characters. In this case, the number of non-blank characters (those representing the data item itself) may not exceed 3 characters. Thus, if d>w,d–w leading blanks will be inserted.

Output Examples

INTERNAL VALUE	SPECIFICATIONS	EXTERNAL STRING
-123.4567	U	-123.4567
789	U	789
1.5@@275	U10	1.5D+275
1234567	U5	1.2+6
1	U10.4	bbb1
123.456	U10.4	123.456
1	U5.8	bbbbbbb1
123.456	U5.8	bbb123.5

V Format

The V format specification allows a variable editing phrase letter to be supplied at run-time. When V appears in a format specification list, the next list element is accessed to furnish the editing letter. Legal list elements are of type **ALPHA, REAL, INTEGER, DOUBLE, BOOLEAN** or **POINTER**. The rightmost character of the list element (only the first word of a double is considered) is used to supply the editing letter. The editing letter extracted from the list element will be a 6-bit character if the default character size is **BCL**; otherwise, an 8-bit character is extracted. If the list element is a <*pointer expression*>, the first character of the designated string is used as the editing letter.

Example:

```
•
•
•
REAL A,B;
DOUBLE D;

FORMAT FMT1(V8.2),
       FMT2(2V*),
       FMT3(*V*.*);
•
•
•
READ(KARD,FMT1, "R", A);
B:=4"C1";
WRITE(LINE,FMT2, B, 6, A, D);
D:=DOUBLE(4"C5",0);
READ(KARD,FMT3, 2, D, 10, 4, A, B);
•
•
•
```

In the above program,

FMT1 evaluates to R8.2 applied to list element A,
FMT2 evaluates to 2A6 applied to list elements A and D,
FMT3 evaluates to 2E10.4 applied to list elements A and B.

X Format

On input, w characters are skipped. On output, w blanks are inserted. No *<list element>* corresponds to this editing letter.

Z Format

The general format specification Zw.d is a generalized floating point conversion which may be used with list elements of type **ALPHA, REAL, INTEGER, DOUBLE** or **BOOLEAN.** This specification is interpreted as D,E,F,I or L format, depending upon the type and magnitude of the value of the list element.

Input

On input, the Zw.d specification is the same as D, E or F formats for **ALPHA, REAL** and **DOUBLE** list elements. For **INTEGER** list elements, Z functions like Iw, and for **BOOLEAN** list elements, Z functions like Lw.

Output

The output string will have a length of w characters, regardless of the value being read or written. For **BOOLEAN** list elements, Lw is used. For **INTEGER** list elements, Iw is used. For **ALPHA, REAL** or **DOUBLE** list elements, a D, E or F format representation of the list element's value is produced according to the following criteria: If V is the absolute value of the list element, then for K=0,1,2,...,d, if $10^{d-K-1} \leqslant V \leqslant 10^{d-K}$, then formats F(w-4) . (d-K), X4 are used. If V< . 1 or $V \leqslant 10^{d}$, then Ew.d is used. In other words, Zw.d implies "output d significant digits".

Output Examples

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
1.23@@250	Z12.6	1.230000+250
1	Z5.1	bbbb1
12345	Z5.1	12345
12	Z8.7	bbbbbb12
12345.678	Z10.4	1.2346E+04
12	Z10.4	bbbbbbbb12
12345678	Z6	******
1234	Z6	bb1234
1 (BOOLEAN)	Z3	TRU

FORWARD REFERENCE DECLARATION

Syntax

<forward reference declaration> ::= *<forward interrupt declaration>* |
<forward procedure declaration> |
<forward switch declaration>
<forward interrupt declaration> ::= **INTERRUPT** *<interrupt identifier>* ; **FORWARD**
<forward procedure declaration> ::= *<procedure type>* **PROCEDURE** *<procedure heading>* ;
FORWARD
<forward switch declaration> ::= **SWITCH** *<switch label identifier>* **FORWARD**

Examples

```
SWITCH SELECT FORWARD
INTEGER PROCEDURE SUM (A,B,C);
    VALUE A,B,C;
    INTEGER A,B,C;
    FORWARD
```

Semantics

Before a procedure, switch, or interrupt can be used in a program, it must be declared. However, consider the following situation: in the body of procedure #1, a reference is made to procedure #2. Likewise, within the body of procedure #2, a call is made on procedure #1. Regardless of which procedure is declared first, its body contains a reference to an undeclared entity. A similar situation can be constructed with two switches, because these constructs also have the power of recursion.

To enable the **ALGOL** compiler to handle situations of this nature, the *<forward reference declaration>* is necessary. Therefore, in the example given above, the body of procedure #1 might be a *<block>* containing the *<declaration>* **PROCEDURE TWO; FORWARD.** Later in this *<block>*, procedure #2 is called and the compiler recognizes it. Finally, at some later point in the program, procedure #2 is declared in full.

INTEGER DECLARATION

Syntax

<integer declaration> ::= *<local or own>* **INTEGER** *<identifier list>* |
INTEGER *<equation list>*

Examples

INTEGER INTGR
INTEGER COUNT, VAL, NOEXPONENT
INTEGER INT=BOOL, CAL, NUM=REEL

Semantics

An *<integer declaration>* is used to declare *<simple variable>s* which can be used as integer values, that is, an arithmetic value that is maintained as a value with an exponent of zero.

The *<local or own>* portion of the *<integer declaration>* indicates whether the value of the specified *<simple variable>* is to be retained upon exit from the *<block>* in which it is declared. A *<simple variable>* declared to be **OWN** will retain its value when the program exits from the associated *<block>*, and that "old" value will be the contents of the *<simple variable>* when the associated *<block>* is re-entered.

Upon entry to a *<block>* containing *<simple variable>s*, the normal content of a non-**OWN** *<simple variable>* is a zero (0); i.e., a 48-bit word with all bits off. To be truly compatible with **ALGOL-60**, a programmer would explicitly zero the *<simple variable>s* with appropriate *<assignment statement>s.*

The *<equation list>* allows address equation among real, integer, and Boolean variables only. An *<identifier>* may only be address-equated to a previously declared local *<identifier>* or to an *<identifier>* global to the block in which it is declared.

Pragmatics

After an arithmetic calculation, the resulting value is integerized and then stored into the *<simple variable>*, in contrast to a real *<simple variable>* which is stored "as is."

Appendix B contains additional information on the internal structure of an integer *<simple variable>* as implemented on the **B 7000/B 6000** Information Processing System.

INTERRUPT DECLARATION

Syntax

<interrupt declaration> ::= **INTERRUPT** *<interrupt identifier>* ; *<unlabeled statement>*

<interrupt identifier> ::= *<identifier>*

Example

```
INTERRUPT ERR; GO TO ABORT
INTERRUPT I1;
    BEGIN
    . . .
    . . .
    END
```

Semantics

The *<interrupt declaration>* provides a means of forcing a process to depart from its current point of control and execute the *<unlabeled statement>* associated with the *<interrupt declaration>*. The process then normally returns to its previous point of control when the program "falls out the end" of the *<unlabeled statement>*. However, this would not be the case if a *<go to statement>* is executed within the *<unlabeled statement>* and the specified *<label>* is outside of the *<unlabeled statement>*.

An interrupt must be enabled (refer to *<enable statement>*) and attached to an event by an *<attach statement>* before it can have any effect. The *<disable statement>* can temporarily render the associated interrupt ineffective.

Pragmatics

An *<interrupt declaration>* can be thought of as describing an *<unlabeled statement>* (which can also be a *<block>*) which is automatically entered upon the occurrence (CAUSE) of an event. The MCP ensures when a program is executing the *<unlabeled statement>*, all other interrupts are queued until the program exits from the *<unlabeled statement>*.

LABEL DECLARATION

Syntax

<label declaration> ::= **LABEL** *<label identifier list>*
<label identifier list> ::= *<label identifier>* |
<label identifier list> , *<label identifier>*
<label identifier> ::= *<identifier>*

Examples

LABEL START
LABEL ENTER,EXIT,START,LOOP

Semantics

A *<label declaration>* declares each identifier in its *<identifier list>* as a *<label identifier>*. A *<label identifier>* must appear in a *<label declaration>* in the head of the innermost block in which it is used to label a statement. If any *<statement>* in a *<procedure body>* is labeled, the declaration of this label must appear within the *<procedure body>*.

LIST DECLARATION

Syntax

<list declaration> ::= **LIST** *<list part list>*
<list part list> ::= *<list part>* | *<list part list>* , *<list part>*
<list part> ::= *<list identifier>* (*<list segment>*)
<list identifier> ::= *<identifier>*
<list segment> ::= *<list element>* | *<list segment>* , *<list element>*
<list element> ::= *<unconditional list element>* |
 * *<unconditional list element>* |
 <conditional list element> |
 * *<conditional list element>*

<unconditional list element> ::= *<simple arithmetic expression>* |
 <simple Boolean> | *<pointer expression>* |
 <pointer expression> **FOR** *<arithmetic expression>* |
 <array row> | [*<list segment>*] | **DO** *<list element>* **UNTIL**
 <Boolean expression> |
 <iteration clause>
 <unconditional list element> |
 <if clause> *<unconditional list element>* **ELSE**
 <unconditional list element> |
 CASE *<arithmetic expression>* **OF** (*<list segment>*)

<iteration clause> ::= **FOR** *<variable>* := *<for list>* **DO** |
 THRU *<arithmetic expression>* **DO** |
 WHILE *<Boolean expression>* **DO**

<conditional list element> ::= *<if clause>* *<list element>* |
 <iteration clause> *<conditional list element>* |
 <if clause> *<unconditional list element>* **ELSE**
 <conditional list element> |
 DO *<list element>* **UNTIL** *<Boolean expression>* |
 CASE *<arithmetic expression>* **OF** (*<list element>*)

Examples

```
LIST L1 (X,Y,A,[J], FOR I := P STEP 1 UNTIL 5 DO B [I])
LIST ANSWERS (P + Q,Z,SQRT (R)), RESULTS (X1,X2,X3,X4/2)
LIST LIST3 (FOR I := 0 STEP 1 UNTIL 10 DO FOR J := 0, 3, 6
    DO A[I,J])
LIST L4 (B AND C, NOT AB1, IF X = 0 THEN R1 ELSE R2)
LIST RESULTS (FOR I := 1 STEP 1 UNTIL N DO [A[I], FOR J :=1
    STEP 1 UNTIL K DO [B[I,J], C[J]]])
```

Semantics

A *<list declaration>* associates an ordered set of *<list element>s* with a *<list identifier>*. A *<list identifier>* is usually used in conjunction with a *<format identifier>* within a *<read statement>* or *<write statement>* to indicate which entities are to be associated with the corresponding *<editing phrase>s* of the specified format. Although the syntax of the *<read statement>* and *<write statement>* allows the entities to be listed within the statement itself, a *<list declaration>* provides a more convenient means of grouping the entities to be used. *<list element>s* can be either conditional or unconditional.

<unconditional list element>s

<unconditional list element>s are the usual entities found in *<list segment>s*. Essentially they are built from arithmetic primaries, Boolean primaries, pointer primaries, and array rows.

<pointer expression> **FOR** *<arithmetic expression>*

<pointer expression> **FOR** *<arithmetic expression>* allows the user to specify the amount of the string, to which the pointer points, to be used as a list element. Thus, if P points at string "ABCDEFGHIJKL", P FOR 3 refers to the substring "ABC".

Asterisks

Asterisks (*) prefixed to a list element only have meaning for free-field output (they are ignored for other I/O). The asterisk prefixed to a list element will cause, under the control of free-field output, the text of the list element to be output just prior to the edited value of the list element, with an equal sign (=) inserted between the two. If the list element is a string under control of any other I/O, the prefixed asterisk is ignored.

MONITOR DECLARATION

Syntax

<monitor declaration> ::= **MONITOR** *<monitor part list>*
<monitor part list> ::= *<monitor part>* |
<monitor part list> , *<monitor part>*
<monitor part> ::= *<file or procedure identifier>* (*<monitor list>*)
<file or procedure identifier> ::= *<file identifier>* |
<procedure identifier>
<monitor list> ::= *<monitor element>* |
<monitor list> , *<monitor element>*
<monitor element> ::= *<simple variable>* |
<label identifier> |
<array identifier>

Examples

```
MONITOR FYLE(A)
MONITOR PRNTR(X,LBL,ARAY)
MONITOR MONPROC(VAL,INDX,INFO)
```

Semantics

The diagnostic *<monitor declaration>* causes all subsequent occurrences of assignments of the form *<monitor element>* := to produce monitoring action during execution of the program. Each time an *<identifier>* included in the *<monitor list>* is used in one of the ways described in the following paragraphs, the *<identifier>* and its current value are written on the file or passed as parameters to the procedure specified in the *<monitor declaration>*. In particular, the monitor action does not occur for assignments within procedures that are declared before the *<monitor declaration>* is encountered, nor does monitoring of a variable in the *<monitor list>* occur if this *<identifier>* is used as a call-by-name *<actual parameter>* to a procedure that modifies the value(s) of its *<formal parameter>s*.

Pragmatics

The diagnostic information produced depends on the form(s) of the *<monitor element>s*. When the **$LINEINFO** compiler option is **SET**, and a *<file identifier>* is specified as the *<monitor part>*, the stack number, an @ sign, a segment address, and a sequence number are printed in front of the symbolic name of the *<monitor element>*. For example, 0143 @ 003:0003:4 (00007000). Diagnostic information is provided as follows by the specified *<monitor element>s:*

a. When the *<monitor element>* is a *<simple variable>*, the symbolic name and the before and after values of the *<simple variable>* are printed. For example, B =0:=13. The controlled variable in a *<for statement>* cannot be monitored.
b. When the *<monitor element>* is a *<label identifier>*, the symbolic name of the label is shown. For example, **LABEL L.**
c. If the *<monitor element>* is an *<array identifier>*, the symbolic name of the array and the before and after values of the specified *<array element>* are printed. For example, **ARAY [12] =0:=12.**

The *<file identifier>* cannot be a file-valued task attribute.

The *<monitor part>* of the form *<procedure identifier>* (*<monitor list>*) produces the following information when the applicable restrictions are observed. Note that printing of the *<monitor element>* is not automatic when a *<procedure identifier>* is used. Printing must be performed by the procedure. Also, the monitored procedure performs the specified operations depending on the values passed to it.

When the *<simple variable>* form is used, the format of the monitoring procedure must be in the following general form:

REAL PROCEDURE MON (NAME,VAL,SPELL);

The procedure must be of the same *<monitor list>*. The procedure must have three arguments:

a. The first parameter **(NAME)** is the name of the *<monitor element>*; that is, the first parameter is call-by-name parameter of the same *<type>* as the *<monitor element>*. This argument **(NAME)** is normally used to store the value of the second argument **(VAL)**.
b. The second parameter **(VAL)** is also of the same *<type>*. But, it is a call-by-value parameters and contains the value to be assigned to the *<monitor element>*.
c. The third and last parameter **(SPELL)** must be a call-by-value **ALPHA** variable. It contains the name of the *<monitor element>* as a string of characters. Only the first six characters of the symbolic name are passed into this *<formal parameter>*. If the symbolic name is less than six characters, the symbolic name is left-justified and trailing blanks are added, up to six characters.

If the *<monitor element>* is to be assigned a value, it must be done by the monitoring procedure. The value returned by the procedure can then be used, for example, in evaluating the remainder of an *<expression>* in which the assignment is imbedded. For example, note that in the succeeding example under *<array identifiers>*, the assignment statement "**NAME:=MON:=VAL;**" allows the subsequent use of the value assigned to the *<monitor element>*.

When the *<label identifier>* form of the *<monitor element>* is used, the format of the monitoring procedure must be in the following general format:

PROCEDURE MON (SPELL);

The procedure must be untyped. It must have only one parameter. This parameter will contain the first six characters of the symbolic name. If the symbolic name is less than six characters, the symbolic name is left-justified and trailing blanks are added, up to six characters. For example, the monitoring procedure could compare the symbolic names in the *<monitor list>* in order to identify a particular label. The spelling of the labels follows the same rule as described under the *<simple variable>* form.

When the *<monitor list>* is of the form *<array identifier>*, the format of the monitoring procedure must be in the following general format:

REAL PROCEDURE MON (D_1 . . . D_n,NAME,VAL,SPELL);

The array to be monitored must have the same number of dimensions as the monitoring procedure. In other words, the first D_1 . . . D_n parameters of the procedure must correspond to the dimensions of the subscripted array variable. Each dimension parameter is a call-by-value integer. The last three parameters are the same as in the *<simple variable>* case. Notice that formal parameter **VAL** is a *<simple variable>*.

The value normally returned by the procedure is the value used to evaluate the remainder of the *<expression>*, if any.

The following procedure could be used to monitor a two-dimensional array so that the values in the array never become negative.

```
REAL PROCEDURE MON (D1, D2, NAME, VAL, SPELL);
VALUE D1, D2, VAL, SPELL;
REAL NAME, VAL;
ALPHA SPELL;
INTEGER D1, D2;
BEGIN
IF VAL  < 0 THEN GO TO ERROREXIT; % "BAD-GO-TO"
NAME:=MON:=VAL; % RETURN VALUE IN CASE OF FURTHER USE
END;
```

The occurrence of the *<statement>* **B:=A[I,J] :=4;** where A is monitored by **MON**, is equivalent to the *<statement>* **B:=MON(I,J,A[I,J],4,"A");**

An array may not be monitored if it is in the *<list>* part of a *<read statement>* or *<write statement>*.

PICTURE DECLARATION

Syntax

<picture declaration> ::= **PICTURE** *<picture part list>*
<picture part list> ::= *<picture part>* |
<picture part list> , *<picture part>*
<picture part> ::= *<picture identifier>* (*<picture>*)
<picture identifier> ::= *<identifier>*
<picture> ::= *<picture symbol>* |
<picture> *<picture symbol>*
<picture symbol> ::= *<string>* |
<picture character> *<repeat part value>* |
<control character> |
<introduction> |
<picture skip> *<repeat part value>* |
<single picture character>
<picture character> ::= **A** | **D** | **E** | **F** | **I** | **R** | **X** | **Z** | **9**
<repeat part value> ::= *<empty>* |
(*<unsigned integer>*)
<control character> ::= **Q** | :
<introduction> ::= *<introduction code>* *<new character>* |
4 *<introduction code>* *<hexadecimal character>*
<introduction code> ::= **B** | **C** | **M** | **N** | **P** | **U**
<new character> ::= <EBCDIC character> | "
<picture skip> ::= ⟩ | ⟨
<single picture character> ::= **J** | **S**

Examples

PICTURE Z9S (ZZZZ9)
PICTURE PF ("FIRST IS" X(1)A(1)⟩(10)X(1)"LATER IS" X(1)A(1)I(3)⟩(11)A(3))
PICTURE USECS (ZZZI999999)
PICTURE TIMENOW (" "N:9(2)I9(2)I9(2))

Semantics

The *<picture declaration>* provides a means of performing generalized character editing. Pictures are used in *<replace statement>s.* The following editing operations can be performed:

a. Unconditional character moves.
b. Move characters with leading zero editing.
c. Move characters with leading zero editing and floating character insertion.
d. Move characters with conditional character insertion.
e. Move characters with unconditional character insertion.
f. Move numeric part of characters only.
g. Skip source characters, forward and reverse.
h. Skip destination characters forward.
i. Insert overpunch sign on the previous character.

A *<picture>* consists of a named string of editing symbols that are enclosed in parentheses. The picture editing symbols listed below can be combined in any order to perform a wide range of editing functions.

<introduction code>s

The output characters listed below are assumed for the *<introduction code>s.* Another character can be substituted for the assumed character by the use of the *<introduction>* phrase, as defined in the syntax. The two hexadecimal characters are assumed to represent a single <EBCDIC *character*>.

OUTPUT CHARACTER	INTRODUCTION CODE	NORMAL USE
space (blank)	B	Replacement of leading zeros
,	C	Conditional insert character
–	M	Character insertion if minus
.	N	Unconditional insert character
+	P	Character insertion if plus
$	U	Floating character insertion

<control character>s

The control characters shown below cause the following action:

CHARACTER	ACTION
Q	Inserts an overpunch sign in the preceding character position.
:	Re-initiates leading zero replacement.

<single picture character>s

The *<single picture character>s* perform the following action:

CHARACTER	ACTION
J	If a move with float (E or F) has not inserted a float character, the float is terminated and the U character is inserted. Otherwise, no operation is performed.
S	A single P character is inserted if the sign is plus; otherwise, a single M character is inserted.

<picture character>s

The *<picture character>s* listed below perform the following action:

CHARACTER	ACTION
A	Moves the number of characters specified by the *<repeat part value>*.
D	If an E or F float has not ended, the B character is inserted. Otherwise, the C character is inserted.

CHARACTER	ACTION
E	Moves the numeric part only for the number of characters specified by the *<repeat part value>*. Suppresses leading zeros by substituting the B character. If the sign is plus, a P character is inserted in front of the first non-zero number. Otherwise, an M character is inserted and the float is ended.
F	A move numeric is performed with leading zeros replaced by the B character. A U character is inserted in front of the first non-zero number, and the float action is ended.
I	The N character is inserted unconditionally.
R	If an E or F float has not ended, the P character is inserted. Otherwise, the M character is inserted.
X	The destination pointer is skipped forward by the number of characters specified in the *<repeat part value>*.
Z	A move numeric is performed with leading zeros replaced by blanks.
9	Moves the numeric part only of the number of characters specified by the repeat field.

<picture skip>

The *<picture skip>* characters perform the following action:

CHARACTER	ACTION
〈	The source pointer is skipped in reverse (to the left) by the number of characters specified by the *<repeat part value>*.
〉	The source pointer is skipped forward (to the right) by the number of characters specified by the *<repeat part value>*.

Pragmatics

One value array (also called an "edit table") is generated for each *<picture declaration>* and therefore it would generally be wise to collect all pictures under a single *<picture declaration>*.

POINTER DECLARATION

Syntax

<pointer declaration> ::= **POINTER** *<pointer identifier list>*
<pointer identifier list> ::= *<pointer identifier>* |
<pointer identifier list> , *<pointer identifier>*
<pointer identifier> ::= *<identifier>*

Examples

POINTER PTR
POINTER PTS,PTD,SORCE,DEST

Semantics

A pointer represents the relative address of a character position with respect to the beginning of a one-dimensional array or an *<array row>*. Thus, it is said to "point" to a character position. The *<pointer declaration>* establishes each *<identifier>* in the pointer list as a *<pointer identifier>*.

Pragmatics

Pointers are initialized via a *<pointer assignment>* statement. Any attempt to use a pointer prior to its initialization will result in an **INVALIDOP** error.

A pointer should not be initialized to point into an *<array row>* which is "up-level." Stated in another way, the *<pointer declaration>* should be at the same or higher lexicographical level as the referenced declaration of the *<array row>*; it should not be lower. If it is lower, total system failure can occur.

PROCEDURE DECLARATION

Syntax

```
<procedure declaration> ::= <procedure type> PROCEDURE
                                 <procedure heading>;<procedure body>
<procedure type> ::= <empty> |
                     <type>
<procedure heading> ::= <procedure identifier> <formal parameter part>
<procedure identifier> ::= <identifier>
<formal parameter part> ::= <empty> |
                            ( <formal parameter list> ) ; <value part> <specification part>
<formal parameter list> ::= <formal parameter> |
                            <formal parameter list> <parameter delimiter> <formal parameter>
<formal parameter> ::= <identifier>
<value part> ::= <empty> |
                 VALUE <identifier list> ;
<specification part> ::= <specification> |
                         <specification part> ; <specification>
<specification> ::= <specifier> <identifier list> | <procedure type> PROCEDURE <identifier list>
                         <formal parameter specifier> |
                    <array specification>
<specifier> ::= <direct specifier> FILE |
                <direct specifier> SWITCH FILE |
                EVENT|
                FORMAT |
                LABEL |
                LIST |
                PICTURE |
                POINTER |
                SWITCH |
                SWITCH FORMAT |
                SWITCH LIST |
                TASK |
                <type>
<formal parameter specifier> ::= <empty> |
                                 ( ) ; FORMAL |
                                 <value part> <specification part> ; FORMAL
<array specification> ::= <direct specifier> <array type> <array specifier list>
<array type> ::= <array class> |
                 EVENT |
                 TASK
<array specifier list> ::= <array specifier> |
                           <array specifier list> , <array specifier>
<array specifier> ::= <array identifier list> [ <lower bound list> ]
<lower bound list> ::= <specified lower bound> |
                       <lower bound list> , <specified lower bound>
<specified lower bound> ::= <integer> | *
<procedure body> ::= <unlabeled statement> |
                     EXTERNAL
```

Examples

```
PROCEDURE SIMPL; X := X + 1
PROCEDURE TUFFER (PARAM);
    VALUE PARAM;
    REAL PARAM;
    X := X + PARAM
REAL PROCEDURE RESULT (PARAM,FYLEIN);
    REAL PARAM;
    FILE FYLEIN;
    BEGIN
      .
      .
      .
    RESULT := X + PARAM;
      .
      .
      .
    END
BOOLEAN PROCEDURE MATCH (A,B,C);
    VALUE A,B,C;
    INTEGER A,B,C;
    MATCH := A=B OR A=C OR B=C
DOUBLE PROCEDURE MUCHO (DDBL1,DBL2,BOOL);
    VALUE DBL2,BOOL;
    DOUBLE DBL2;
    BOOLEAN BOOL;
    BEGIN
    REAL LOCALX,LOCALY;
      .
      .
      .
    MUCHO := DOUBLE (LOCALX,LOCALY);
    END OF MUCHO
PROCEDURE FURTHERON;
    FORWARD
INTEGER PROCEDURE BOWNDIN (P1,P2,P3,P4);
    VALUE P2, P4;
    POINTER P1;
    REAL P2, P3;
    FILE P4;
    EXTERNAL
```

Semantics

A *<procedure declaration>* defines the *<procedure identifier>* as the name of a procedure.

A procedure becomes a "function" by preceding the word **PROCEDURE** with a *<type>* and by assigning a value or result to the procedure somewhere within the *<procedure body>*. (Refer to **EXAMPLES: RESULT, MATCH,** and **MUCHO.**) This kind of procedure is more commonly referred to as a "typed procedure" and is known to return a result. Note that a typed procedure can be used either as a

<statement> or as an *<expression>*. When used as a *<statement>*, the returned result is automatically discarded.

The purpose of the *<formal parameter part>* is to list the item(s) which will be "passed in" as parameters when the procedure is invoked. As can be seen from the syntax, a *<formal parameter part>* is optional. If it is supplied, a *<value part>* and *<specification part>* are then required.

The *<value part>* specifies which *<formal parameter>s* are to be "called by value." When a *<formal parameter>* is called by value, the *<formal parameter>* is set to the value of the corresponding *<actual parameter>*. Thereafter, the *<formal parameter>* is handled as a *<variable>* that is local to the *<procedure body>*. That is, any change of value of the *<variable>* will not ramify outside the *<procedure body>*.

NOTE

Only arithmetic, Boolean, and pointer expressions may be given as *<actual parameter>s* to be called-by-value. These expressions will be evaluated once, before entry into the *<procedure body>*.

<formal parameter>s not in the *<value part>* are "called-by-name." This means that wherever a *<formal parameter>* called-by-name appears in the *<procedure body>*, the *<formal parameter>* is replaced by the *<actual parameter>* and not its value. A call-by-name *<formal parameter>* is effectively global to the *<procedure body>*, since any change in its value within the *<procedure body>* is effected outside the *<procedure body>* on the corresponding *<actual parameter>*.

It is possible to pass an *<arithmetic expression>* as an *<actual parameter>* to a procedure where it has an arithmetic variable specified as call-by-name. This situation results in a "thunk" (also called "accidental entry" or "spontaneous entry") into a compiler-generated typed procedure which is in fact the calculation of the *<arithmetic expression>*. Note that this can be time-consuming if the arithmetic variable is repeatedly referenced. Furthermore, an invalid operand interrupt error will occur if an attempt is made to store into that item.

Every *<formal parameter>* must appear in the *<specification part>*.

The *<array specification>* must be provided for every array passed into the procedure. The primary purpose of the *<array specification>* is to specify the number of dimensions in the passed array and to indicate the *<specified lower bound>* as desired within the *<procedure body>*.

A *<specified lower bound>* which is an *<integer>* denotes that the corresponding dimension of the *<actual parameter>* has a declared *<lower bound>* equal to this value. If an "*" is used as a *<specified lower bound>*, it indicates that the corresponding dimension of the *<actual parameter>* has a declared *<lower bound>* that may vary in value.

The **EXAMPLES** show how the *<procedure body>* of a procedure can vary in complexity from a basic *<unlabeled statement>* to a *<block>*.

PROCEDURE FURTHERON shows the means of declaring that a procedure exists "later" in the program. (More on this is said under *<forward procedure declaration>*.)

The last **EXAMPLE** illustrates the method of specifying a procedure that will be "bound in" as compared to "compiled in" to the program. An attempt to reference the procedure that has not been bound in will result in a run-time error.

Pragmatics

Procedures may be called recursively; i.e., inside the *<procedure body>*, a procedure may invoke itself.

For purposes of efficiency, it is advisable to call-by-value as many *<formal parameter>s* as possible. Secondly, the *<specified lower bound>s* should have a value of 0 for the *<integer>*.

Array rows that are passed by name as actual parameters to procedures will have their subscripts evaluated at the time of the procedure call, rather than at the time the corresponding formal array is referenced.

FORMAL causes the compiler to generate more efficient code when passing procedures as parameters. That is, when procedures are declared **FORMAL**, the compiler checks the parameters at compile-time; otherwise, the parameters are checked at run-time.

REAL DECLARATION

Syntax

<real declaration> ::= *<local or own>* **REAL** *<identifier list>* |
REAL *<equation list>*

Examples

REAL REEL
REAL INDX, X, Y, TOTAL
REAL CALC=BOOL, INDX, VALV=INTGR

Semantics

A *<real declaration>* is used to declare *<simple variable>s* which can be used as real values, that is, an arithmetic value which may or may not have an exponent.

The *<local or own>* portion of the *<real declaration>* indicates whether the value of the specified *<simple variable>* is to be retained upon exit from the *<block>* in which it is declared. A *<simple variable>* declared to be **OWN** will retain its value when the program exits from the associated *<block>*, and that "old" value will be the contents of the *<simple variable>* when the associated *<block>* is re-entered.

Upon entry to a *<block>* containing *<simple variable>s*, the normal content of a non-**OWN** *<simple variable>* is a zero (0); i.e., a 48-bit word with all bits off. To be truly compatible with **ALGOL-60**, a programmer would explicitly zero the *<simple variable>s* with appropriate *<assignment statement>s.*

The *<equation list>* allows address equation among real, integer, and Boolean variables only. An *<identifier>* may only be address-equated to a previously declared local *<identifier>* or to an *<identifier>* global to the block in which it is declared.

Pragmatics

After an arithmetic calculation, the resulting value is stored "as is" into the *<simple variable>*, in contrast to an integer *<simple variable>*.

Appendix B contains additional information on the internal structure of a real *<simple variable>* as implemented on the **B 7000/B 6000** Information Processing System.

SWITCH DECLARATION

Syntax

<switch declaration> ::= *<switch file declaration>* |
<switch format declaration> |
<switch label declaration> |
<switch list declaration>

Examples

```
SWITCH FILE SWFILE := . . .
SWITCH FORMAT SWFORM := . . .
SWITCH SWUTCH := . . .
SWITCH LIST SWLIST := . . .
```

<switch declaration>s and their corresponding designators provides an efficient means of dynamically selecting one of many alternative entities of similar kind at a particular point during execution. The entity selected by the use of a switch designator is determined by first evaluating its *<subscript>*. The value of this *<subscript>* is then integerized by rounding, if not already integral, and is used as an index into the list specified in the corresponding *<switch declaration>*.

With the exception of switch labels, the N elements in the list are numbered from 0 to N-1 in their order of appearance, and if the index value lies outside this range, an **INVALID INDEX** error occurs. The range for switch labels is 1 to N. (Refer to *<switch label declaration>*.)

SWITCH FILE DECLARATION

Syntax

<switch file declaration> ::= *<direct specifier>* **SWITCH FILE**
<switch file identifier> := *<switch file list>*
<switch file identifier> ::= *<identifier>*
<switch file list> ::= *<file designator>* |
<switch file list> , *<file designator>*
<file designator> ::= *<file identifier>* |
<switch file identifier> [*<subscript>*] |
<direct file identifier> |
<direct switch file identifier> [*<subscript>*]
<subscript> ::= *<arithmetic expression>*
<direct switch file identifier> ::= *<identifier>*
<direct file identifier> ::= *<identifier>*

Examples

```
SWITCH FILE SWHTAPE := TAPE1, TAPE2, TAPE3;
SWITCH FILE SWHUNIT := CARDOUT, TAPEOUT, PRINT;
```

Semantics

A *<switch file declaration>* associates an *<identifier>* with the *<file designator>s* in the *<switch file list>*. Each *<file designator>* in a *<switch file list>* must reference a previously declared file.

Associated with each of the *<file designator>s* in the *<switch file list>* is an integer reference. The references are 0, 1, 2, . . . , obtained by counting the identifiers from left-to-right. This integer indicates the position of the *<file designator>* in the list. The *<file designator>s* are referenced, according to position, by switch *<file designator>s.*

If the switch *<file designator>* yields a value which is outside the range of the *<switch file list>*, the file so referenced is undefined, and an **INVALID INDEX** error occurs.

Restriction

The *<file designator>s* of a *<switch file declaration>* must all be the same; i.e., if non-**DIRECT**, then all members must be non-**DIRECT**. Normal and **DIRECT** files cannot be mixed in a *<switch file declaration>*.

SWITCH FORMAT DECLARATION

Syntax

```
<switch format declaration> ::= SWITCH FORMAT <switch format identifier> := <switch format list>
<switch format identifier> ::= <identifier>
<switch format list> ::= <switch format segment> |
                         <switch format list> , <switch format segment>
<switch format segment> ::= <format designator> |
                            ( <editing specifications> ) |
                            〈 <editing specifications> 〉
<format designator> ::= <format identifier> |
                        <switch format identifier> [ <subscript> ]
```

Examples

```
SWITCH FORMAT SF:= (A6, 3I4, I2, X60) , (I4,X2,2I4,3I2),
                   (X78,I2) , (X2);
SWITCH FORMAT SWHFT := XF3  ,  XA3, BAF;
```

Semantics

The *<switch format declaration>* associates a *<switch format identifier>* with the switch format segments in the *<switch format list>*. Associated with each of the N *<switch format segment>s* is an integer value from 0 to N-1, which is obtained by counting the segments as they appear from left-to-right. When the corresponding *<format designator>* occurs, its integerized *<subscript>* selects the associated *<switch format segment>*.

If a switch format designator yields a value which is outside the range of *<switch format list>*, the format so referenced is undefined, and an **INVALID INDEX** error occurs.

A *<simple string>* in a *<switch format declaration>* is always read-only if the *<switch format segment>* in which it appears is of the form (*<editing specifications>*).

SWITCH LABEL DECLARATION

Syntax

<switch label declaration> ::= **SWITCH** *<switch label identifier>* := *<switch label list>*

<switch label identifier> ::= *<identifier>*
<switch label list> ::= *<designational expression>* |
<switch label list> , *<designational expression>*

Examples

```
SWITCH CHOOSEPATH := L1, L2, L3, L4,
    SW1 [3], LAB
SWITCH SELECT := START, ERRORI,
    CHOOSEPATH [I + 2]
```

Semantics

A *<switch label declaration>* declares an *<identifier>* to represent a set of *<designational expression>s* as denoted by the *<switch label list>*. Associated with each *<designational expression>*, in the order in which the *<designational expression>* appears in the *<switch label list>*, is an *<integer>* from 1 to N, where N is the number of *<designational expression>s* in the *<switch label list>*. If the index to the switch is an invalid value (≤ 0 or $>$ N), the instruction attempting to branch to it is not executed, and, control proceeds to the next instruction. (Typically, the next statement would be some form of error handling.)

Note that if a *<designational expression>* occurs within a *<switch label list>*, it could reference itself. For example, if N = 4 in the declaration **SWITCH SW := L1, L2, L3, SW[N] ;**, the *<designational expression>* is referencing itself. If it references itself, a **STACKOVERFLOW** condition occurs.

SWITCH LIST DECLARATION

Syntax

<switch list declaration> ::= **SWITCH LIST** *<switch list identifier>* := *<switch list list>*

<switch list identifier> ::= *<identifier>*
<switch list list> ::= *<list designator>* |
<list designator> , *<switch list list>*
<list designator> ::= *<list identifier>* |
<switch list identifier> [*<subscript>*]

Examples

SWITCH LIST CHOOSEPATH := L1, L2, L3, L4, SW1 [3], LAB
SWITCH LIST SELECT := START, ERRORI, CHOOSEPATH [I+2]

Semantics

A *<switch list declaration>* declares an *<identifier>* to represent a set of *<list designator>s* as denoted by the *<switch list list>*. Associated with each *<list designator>*, in its order of appearance in the *<switch list list>*, is an *<integer>* from 0 to N-1, where N is the number of *<list designator>s* in the *<switch list list>*. If the index to the *<switch list list>* is a value which is outside the range of the *<switch list list>*, the list so referenced is undefined, and an **INVALID INDEX** error occurs.

TASK and TASK ARRAY DECLARATIONS

Syntax

<task declaration> ::= **TASK** *<task identifier list>*
<task identifier list> ::= *<task identifier>* |
<task identifier list> **,** *<task identifier>*
<task identifier> ::= *<identifier>*
<task array declaration> ::= **TASK ARRAY** *<task segment list>*
<task segment list> ::= *<task segment>* |
<task segment list> , *<task segment>*
<task segment> ::= *<task array identifier list>* [*<bound pair list>*]
<task array identifier list> ::= *<task array identifier>* |
<task array identifier list> **,** *<task array identifier>*
<task array identifier> ::= *<identifier>*

Examples

```
TASK TSK
TASK TISKIT, TASKIT
TASK ARRAY TSKS [0:9]
TASK ARRAY PROGENY,CHILDREN [0:LIM]
```

Semantics

When a process or co-routine is invoked, a *<task identifier>* is associated with it. While the process or co-routine remains active, various aspects of the process or co-routine can be altered and/or interrogated via the task attributes. Refer to *<arithmetic task attribute>* and *<Boolean task attribute>*.

A task array can have no more than 15 dimensions.

TRANSLATETABLE DECLARATION

Syntax

<translatetable declaration> ::= **TRANSLATETABLE** *<translatetable list>*
<translatetable list> ::= *<translatetable element>* |
<translatetable list> , *<translatetable element>*
<translatetable element> ::= *<translatetable identifier>* (*<translation list>*)
<translatetable identifier> ::= *<identifier>*
<translation list> ::= *<translation specifier>* |
<translation list> , *<translation specifier>*
<translation specifier> ::= *<source characters>* **TO** *<destination characters>* |
<translatetable identifier>
<source characters> ::= *<string>* |
<character set>
<destination characters> ::= *<string>* |
<character set> |
<special destination character>
<character set> ::= **BCL** | **EBCDIC** | **ASCII** | **HEX**
<special destination character> ::= *<string>*

Examples

TRANSLATETABLE TT1 (BCL TO EBCDIC, 6"+" TO 48 "4E"),
TT2 (4"012345689ABCDEF" TO HEX),
TT3 (8 "(" TO 8"[")
TRANSLATETABLE EXPOSEALFA (EBCDIC TO ".",
"ABCDEFGHIJKLMNOPQRSTUVWXYZ" TO "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
"0123456789" TO "0123456789")

Semantics

A *<translatetable declaration>* defines one or more translate tables that can be used with the *<replace statement>*.

The *<character set>* element is equivalent to a string containing all characters in the specified set, in ascending binary sequence, whose length is equal to the total number of characters in the set.

The scope of a *<string>* is the characters in the *<string>*. The length of a *<string>* is its length in terms of its maximum internal character size.

Each succeeding *<translation specifier>* overrides, within its scope, previous *<translation specifier>s*.

Within a *<translation list>*, all source character sizes must be the same and all destination character sizes must be the same, although the character sizes of the source and destination parts need not be the same.

The length of the *<destination part>* must equal the length of the *<source part>*, unless the *<special*

destination character> is used, or if the <*character set*> is used for both the <*source part*> and the <*destination part*>. If the <*special destination character*> is used, all characters within the scope of the <*source part*> are translated to the <*special destination character*>; this character must be a string whose length is one (1) in terms of its maximum internal character size.

Every translate table has a default base in which all source characters are translated to zero characters (all bits **OFF**). The use of a <*character set*> for both the source and destination parts invokes the standard table from the **MCP** and provides a way of obtaining a legitimate base upon which additional <*translation specifiers*> can be used, if desired, to override certain parts of the standard table. The use of a <*translate table identifier*> as a <*translation specifier*> can also be used to provide a base.

When strings of equal length are used for the source and destination parts, translation is based upon the corresponding positions of the source and destination characters, starting from the left and proceeding to the right.

TRANSLATION TABLE INDEXING

The size of the translation table is determined by the size of the <*source part*> characters (characters to be translated): 4-bit characters, four-word table; 6-bit characters, 16-word table; 7- and 8-bit characters, 64-word table. The translation table is one-dimensional read-only array.

Each word in the translation table (figure 4-1) has the following layout: the low-order 32 bits of each word in the translation table are divided into four 8-bit fields, numbered from left-to-right, 0 to 3. (The high-order 16 bits are zeros.)

When a <*source part*> character is to be translated, the character is divided into two parts: the "word index" and the "field index". The field index consists of the two low-order bits; the word index is the remaining high-order bits.

The word index designates the word in the translation table in which the field index locates the character to be used.

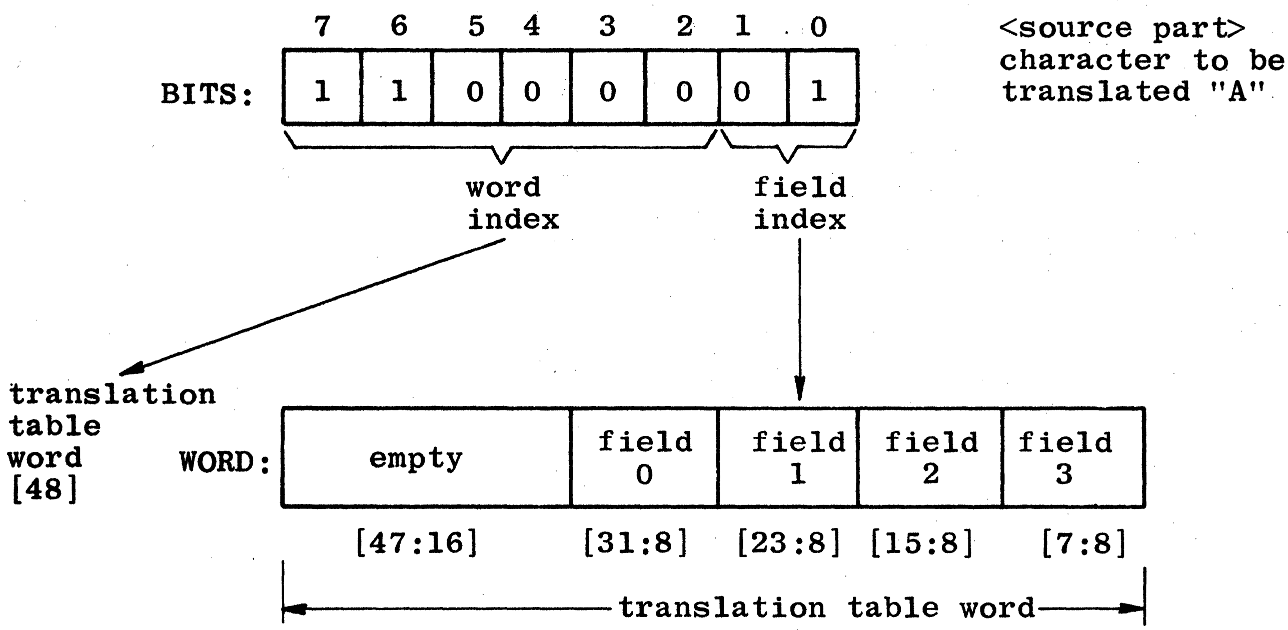


Figure 4–1. Translation Table Indexing

TRUTHSET DECLARATION

Syntax

<truthset declaration> ::= **TRUTHSET** *<truthset list>*
<truthset list> ::= *<truthset element>* |
<truthset list> , *<truthset element>*
<truthset element> ::= *<truthset identifier>* (*<membership expression>*)
<truthset identifier> ::= *<identifier>*
<membership expression> ::= *<membership secondary>* |
<membership expression> *<logical operator>* *<membership secondary>*
<membership secondary> ::= *<membership primary>* |
NOT *<membership primary>*
<membership primary> ::= *<string>* |
<truthset identifier> |
(*<membership expression>*) |
ALPHA | **ALPHA6** | **ALPHA7** | **ALPHA8**

Examples

```
TRUTHSET T(ALPHA)
TRUTHSET Z(ALPHA OR "–")
TRUTHSET NUMBERS ("0123478956")
TRUTHSET LETTERS(ALPHA AND NOT NUMBERS)
TRUTHSET HEXN(4"123"), BCLN(6"123"), ASCN(7"123")
```

Semantics

The *<truthset declaration>* defines one or more truthsets that can be used with the *<scan statement>*, the *<replace statement>*, and with the *<table membership>* Boolean primary.

All membership primaries of a *<membership expression>* must be of the same character type (4, 6, 7, or 8), thereby determining the type of the truthset. The character size of strings is obtained from the maximum internal character size of the string.

The *<membership expression>* is evaluated according to the normal rules of precedence for Boolean operators.

Pragmatics

The *<truthset declaration>* takes a string of characters and builds a "truth table" which allows a programmer to do a truthset test that determines whether a given character is a member of a specified string. The truth table is built from elements that a compiler can completely evaluate at compile-time.

All truthsets declared by a single declaration are made common to a single read-only array. Separate declarations produce separate read-only arrays.

The truthset test references a bit in a read-only array by dividing the binary representation of the character being tested into two parts: the low-order five bits are used as a bit index, and the three high-order bits are used as a word index.

NOTE

> If the source character is 4, 6, or 7 bits, the machine adds high-order zero bits to make an 8-bit character before the "indexing algorithm" is used.

The word index selects a particular word in the read-only array. The bit index is then subracted from 31, and the result is used to reference one of the low-order 32 bits in the selected word. As an algorithm:

ARAY[CHAR.[7:3].[31 – CHAR. [4:5]: 1]]

Finally, the test character is "legitimate" (in the specified string of the declaration) if, and only if, the referenced bit is **ON** (=1).

Figure 4–2 shows that the indexed bit, 13, is **ON**; therefore, the test character is valid.

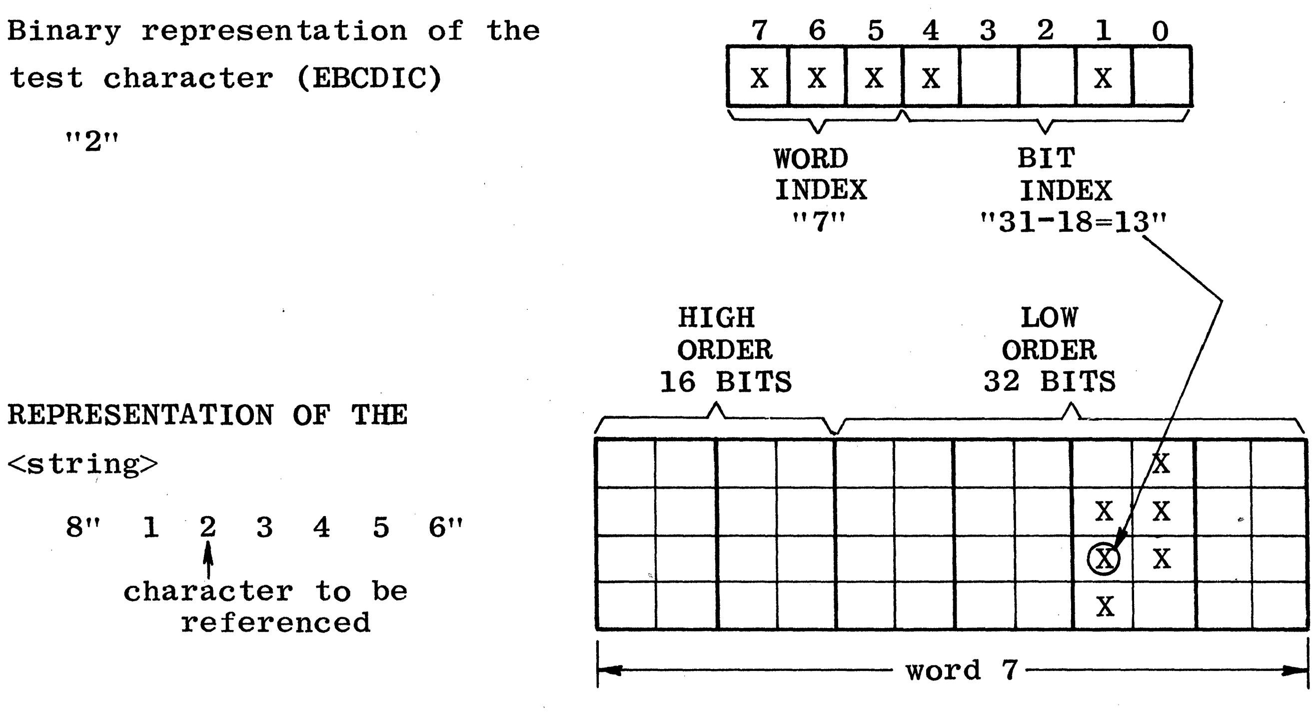


Figure 4–2. Truthset Test

TYPE DECLARATION

Syntax

<type declaration> ::= *<alpha declaration>* |
<Boolean declaration> |
<double declaration> |
<integer declaration> |
<real declaration>

Examples

ALPHA . . .
BOOLEAN . . .
DOUBLE . . .
INTEGER . . .
REAL . . .

Semantics

A *<type declaration>* is used to declare *<simple variable>s* which can be used in a manner appropriate to the specified *<type>*. For example, a variable of *<type>* **BOOLEAN** is normally used in Boolean statements and expressions. Note that the "type transfer function" can be used (as can the *<equation list>* facility) to perform other kinds of operations on a variable than the specified *<type>* of the variable.

Pragmatics

The general use of each *<simple variable>* is as follows:

TYPE	MEANING/DESCRIPTION
ALPHA	Character values; either six, 8-bit characters (normal), or eight, 6-bit characters (**BCL**); treated as *<type>* **REAL.**
BOOLEAN	Logical values; a **TRUE** or **FALSE** test is dependent on the low-order bit (bit 0) of the word; use of the *<partial word part>* allows all 48 bits to be tested and/or manipulated as needed.
DOUBLE	"Double-precision" arithmetic values; a 96-bit entity (carried internally as two adjacent 48-bit words).
INTEGER	Integer arithmetic values; a value which is maintained as a value with an exponent of zero.
REAL	Real arithmetic values; a value which may or may not have an exponent.

Appendix B contains more on the internal structure of each *<simple variable>* as implemented on the **B 7000/B 6000** Information Processing System.

VALUE ARRAY DECLARATION

Syntax

<value array declaration> ::= *<array class>* **VALUE ARRAY** *<value array list>*
<value array list> ::= *<value array segment>* |
<value array list> , *<value array segment>*
<value array segment> ::= *<value array identifier>* (*<constant list>*)
<value array identifier> ::= *<identifier>*
<constant list> ::= *<constant>* |
<constant list> , *<constant>*
<constant> ::= *<unsigned integer>* (*<constant list>*) |
<number> |
<logical value> |
<string> |
<constant expression>
<constant expression> ::= { an expression which can be entirely evaluated by the **ALGOL** compiler at compile-time }

Examples

REAL VALUE ARRAY TEST (3(5, TRUE, "ABC"))
EBCDIC VALUE ARRAY XRAY ("ABCDEFGHIJK")
VALUE ARRAY FOX (1,2,3), CAT (4,5,6)
VALUE ARRAY DOG (2*N+6,7 & 5[3*N:4] & 1[47:1])

Semantics

A *<value array declaration>* defines a read-only one-dimensional array of values.

The *<value array list>* allows the user to specify multiple value arrays of the same *<type>* in one declaration.

The *<unsigned integer>* (*<constant list>*) form of *<constant list>* causes the values within the parentheses to be repeated the number of times specified by the *<unsigned integer>*.

Pragmatics

The comma in the *<constant list>* causes word alignment of the next constant. *<string>s* greater than 48 bits are left-justified with trailing zeros inserted in the word. *<string>s* equal to or less than 48 bits are right-justified with leading zeros inserted in the word. The *<logical value>* and *<number>* *<constant>s* are also right-justified with leading zeros inserted in the word.

The *<constant expression>* builds a 48-bit word from defines, concatenations, arithmetic and Boolean operations, or anything that can be completely evaluated by the compiler at compile-time.

The **MCP** can overlay value arrays more efficiently, since they do not have to be written onto disk when their space in core is relinquished.

5. STATEMENTS

STATEMENT

Syntax

<statement> ::= *<labeled statement>* |
<unlabeled statement>
<labeled statement> ::= *<label identifier>* : *<conditional statement>* |
<label identifier> : *<unconditional statement>*
<unlabeled statement> ::= *<conditional statement>* |
<unconditional statement>

Examples

```
X := 1
LBL: READ ( . . .
IF ALLDONE THEN . . .
NEXTIN: WHILE BOOL DO . . .
```

Semantics

<statement>s are the active elements of an **ALGOL** program. They indicate some type of operation to be performed. *<statement>s* are normally executed sequentially, in the order in which they are written. This sequential flow of execution may be altered by a *<statement>* which indicates its successor to be other than the one which follows it in the program.

As can be seen in the syntax above, *<statement>s* may be labeled or unlabeled. The majority of *<statement>s* in a program are usually unlabeled. Furthermore, the majority of *<unlabeled statement>s* in a program are *<unconditional statement>s.*

This section is arranged in alphabetical order according to the *<statement>*.

ACCEPT STATEMENT

Syntax

<accept statement> ::= **ACCEPT** (*<pointer expression>*)

Example

ACCEPT (POINTER(Z,8))

Semantics

The *<accept statement>* causes **EBCDIC** characters pointed at by the *<pointer expression>* to be displayed on the display console. The maximum number of characters allowed is 430, and the last character must be followed by the **EBCDIC NULL** character (4"00"). The program is then suspended until the appropriate input response is keyed in at a display console. The input is placed in the array row to which the pointer points, and the program continues. The maximum number of input characters allowed is 960.

The *<accept statement>* can be used as a *<Boolean expression>* such that the result is **FALSE** if an input message is not available. If its result is **TRUE**, an input message is available, and it is placed into the array row. In either case, the program is not suspended, but it continues execution.

Pragmatics

The input is placed "left justified" in the array row; i.e., leading blanks are discarded. Following the first non-blank character, the input is placed as-is in the array row and an **EBCDIC NULL** is placed at the end of the input.

ASSIGNMENT STATEMENT

Syntax

<assignment statement> ::= *<arithmetic assignment>* |
<array reference assignment> |
<Boolean assignment> |
<pointer assignment> |
<task assignment>

Examples

A := A + 1
XRAY := ARAY [3,*]
BOOL := FALSE
PTR := POINTER(INARAY,6)
TSK.EXCEPTIONTASK := TSKIT

Semantics

The *<assignment statement>* causes the *<expression>* to the right of the := to be evaluated; the value of the *<expression>* is then assigned to the entity, *<variable>*, or *<partial word part>* on the left.

The action of an *<assignment statement>* is as follows:

a. The *<expression>* following the := is evaluated.
b. The location of the *<variable>* is determined.
c. The resulting value is assigned to the *<variable>* or to the specified part thereof.

Pragmatics

The syntax, examples, semantics, and pragmatics of each form of the *<assignment statement>* are individually discussed in the following pages.

NOTE

The various forms of the *<assignment statement>* are not called *<. . . statement>* because, in general, each of the forms can be used as a form of an *<expression>*. For example, "A := A + 1" would be a *<statement>* if "bracketed" by semicolons (;). However, "**IF A := A + 1 〉** 100" illustrates its use as an *<arithmetic expression>*.

ARITHMETIC ASSIGNMENT

Syntax

```
<arithmetic assignment> ::= <arithmetic variable> <partial word part> := <arithmetic expression> |
                            <arithmetic attribute> := <arithmetic expression> |
                            <type transfer variable> <partial word part> := <arithmetic expression>
<arithmetic variable> ::= <variable>
<variable> ::= <simple variable> |
               <subscripted variable>
<simple variable> ::= <identifier>
<subscripted variable> ::= <array name> [ <subscript list> ]
<array name> ::= <array identifier> |
                 <array reference identifier> |
                 <value array identifier>
<subscript list> ::= <subscript> |
                     <subscript list> , <subscript>
<partial word part> ::= <empty> |
                        [ <left bit> : <number of bits> ]

<left bit> ::= <arithmetic expression>
<number of bits> ::= <arithmetic expression>
<arithmetic attribute> ::= <arithmetic file attribute> |
                           <arithmetic direct array attribute> |
                           <arithmetic task attribute>
<arithmetic file attribute> ::= <file designator> <disk row/copy specifications>
                                .<arithmetic-valued file attribute name>
<disk row/copy specifications> ::= <empty> |
                                   (<row/copy numbers>)
<row/copy numbers> ::= <row number> |
                       <row number> , <copy number>
<row number> ::= <arithmetic expression>
<copy number> ::= <arithmetic expression>
<arithmetic direct array attribute> ::= <direct array row>
                                        .<arithmetic-valued direct array attribute name>
<direct array row> ::= <direct array identifier> |
                       <direct array identifier> [ <row designator> ]
<row designator> ::= * |
                     <row> , *
<row> ::= <arithmetic expression> |
          <row> , <arithmetic expression>
<arithmetic-valued direct array attribute name> ::= IOADDRESS |
                                                    IOCHARACTERS |
                                                    IOCW |
                                                    IOERRORTYPE |
                                                    IOMASK |
                                                    IORECORDNUM |
                                                    IOTIME |
                                                    IOWORDS
```

<arithmetic task attribute> ::=*<task designator>* • *<arithmetic-valued task attribute name>*
<arithmetic-valued task attribute name>::= **CLASS** |
COMPILETYPE |
COREESTIMATE |
DECLAREDPRIORITY |
ELAPSEDTIME |
HISTORY |
INITIATOR |
JOBNUMBER |
MAXCARDS |
MAXIOTIME |
MAXLINES |
MAXPROCTIME |
OPTION |
ORGUNIT |
PROCESSIOTIME |
PROCESSTIME |
RESTART |
STACKNO |
STACKSIZE |
STARTTIME |
STATION |
STATUS |
STOPPOINT |
SUBSPACES |
TARGETTIME |
TASKATTERR |
TASKVALUE |
TYPE
<type transfer variable>::= **REAL** (*<variable>*) | **INTEGER** (*<variable>*) |
BOOLEAN (*<variable>*) | **ALPHA** (*<variable>*) |
DOUBLE (*<variable>*)

Examples

```
VAL:= 7
ARAY [4,5].[30:4] :=X
FYLE.AREAS := 50
FYLE (5). AREAS := 10
DIRARAY'IOCW:=4"1030"
TSK.COREESTIMATE := 10000
```

Semantics

In an *<arithmetic assignment>*, the appropriate implicit *<type>* conversion (**INTEGER, REAL,** or **DOUBLE**) is performed as required.

If there is a difference between the declared *<type>* of the variable to the left of the := and the value to be assigned to it, or if the left-side variables are of different arithmetic *<type>s*, the compiler reconciles the differences, but this can cause a change (rounding to integer) in the value assigned.

The following rules apply:

a. If the left-side is of *<type>***REAL** and the expression value is of *<type>* **INTEGER**, the value is stored unchanged.
b. If the left-side list is of *<type>* **INTEGER** and the expression value is of *<type>* **REAL**, the value is rounded before it is stored.
c. If the left-side list contains variables of different *<type>s*, assignment of the value is executed from right-to-left. If, during this process, a real number is transferred to integer, this integer value is assigned to all the following variables at the left of the integer variable, regardless of their type.

A multiple assignment of an *<arithmetic attribute>* or *<arithmetic variable>* *<partial word part>* is allowed only if it is the first and the only *<arithmetic attribute>* or *<arithmetic variable>* *<partial word part>* within the *<arithmetic assignment>* statement.

Example

The following compile syntactically correct.

```
X.[7:8] : = Y : = 1;
FILE.KIND := Y := 1;
```

The following compile syntactically incorrect.

```
X.[7:8] : = Y. [7:8] : = 1;
FILE.KIND := FILE1.KIND := 2;
```

Pragmatics

An "update replacement" can be specified with an asterisk (*) after the colon equal (:=) by an assignment to an *<arithmetic variable>* whose *<partial word part>* is *<empty>*. For example, "A := *+1; " produces the same results as "A :=A + 1;". Updating a *<subscripted variable>* via this method is more efficient.

<partial word part>

If non-*<empty>*, the *<left bit>* part must specify a bit number of 47 thru 0, inclusive. The *<number of bits>* must specify 48 thru 0, inclusive. If through the use of *<variable>s* a program violates either of these requirements, an **INVALID OP** will occur.

ARRAY REFERENCE ASSIGNMENT

Syntax

<array reference assignemnt>::=<array reference variable>:=<array designator>
<array reference variable>::=<array reference identifier>
<array designator>::=<array name> |
<subarray designator>
<subarray designator>::=<array identifier> [*<subscript part> <subarray part>*] |
<array reference identifier> [*<subscript part> <subarray part>*]
<subscript part>::=<empty>
<subscript list>,
<subarray part>::= * |
<subarray part>, *

Examples

BOOLARAY := REELARAY
EBCDICARAY := INPUTARAY [*]
SUBARAY := BIGARAY [N,*,*]
ARAYROW := MULTIDIMARAY [I,J,K,*]

Semantics

An *<array reference assignment>* is used to generate a "copy descriptor" of an array or portion of an array. Subsequent use of the *<array reference variable>* references the array or portion thereof. (Refer to *<array reference declaration>*.)

The lex level of the *<array designator>* may not be greater than that of the *<array reference variable>*. i.e., the lex level of the *<array reference variable>* may not be global to the *<array designator>*.

If the *<array reference variable>* is declared **DIRECT**, then only **DIRECT** *<array designator>s* may be assigned to it. However, a non-**DIRECT** *<array reference variable>* may be assigned either **DIRECT** or non-**DIRECT** *<array designator>*.

If the number of dimensions of *<array reference variable>* and/or the *<array designator>* are greater than one (1), their *<array class>es* must agree. If they are both single-dimensioned, the *<array designator>* may have any *<array class>*; the generated copy descriptor is modified as necessary to agree with the *<array class>* of the *<array reference variable>*.

Pragmatics

Typical uses of an *<array reference assignment>* would include:

1. a more efficient means of performing arithmetic operations on multi-dimensioned arrays; e.g., extract a particular row and avoid continual multi-indexing back to the same row each time.

2. concurrent but different usages of the same array; e.g., an array which contains either or both **BOOLEAN** and **REAL** information.

BOOLEAN ASSIGNMENT

Syntax

<Boolean assignment> ::= *<Boolean variable> <partial word part>* := *<Boolean expression>* |
<Boolean attribute> := *<Boolean expression>*
<Boolean variable> ::= *<variable>*
<Boolean attribute> ::= *<Boolean file attribute>* |
<Boolean direct array attribute> |
<Boolean task attribute>
<Boolean file attribute> ::= *<file designator> <disk row/copy specifications>*
. *<Boolean-valued file attribute name>*
<Boolean direct array attribute> ::= *<direct array row>* .
<Boolean-valued direct array attribute name>
<Boolean-valued direct array attribute name> ::= **IOCANCEL** |
IOCOMPLETE |
IOEOF |
IOPENDING |
IORESULT
<Boolean task attribute> ::= *<task designator>*.*<Boolean-valued task attribute name>*
<Boolean-valued task attribute name> ::= **LOCKED**

Examples

```
BOOL := TRUE
BOOLARAY [N].[30:1] := Q 〈 VAL
HIGHER := PTR 〉 PTS FOR 6
```

Semantics

A *<Boolean assignment>* is used to store Boolean information (which has either been declared **BOOLEAN** or whose *<type>* has been changed with the **BOOLEAN** type transfer function).

NOTE
The contents stored into a *<Boolean variable>* are the result of a *<Boolean expression>*.

POINTER ASSIGNMENT

Syntax

<pointer assignment> ::= *<pointer variable>* := *<pointer expression>*
<pointer variable> ::= *<pointer identifier>*

Examples

```
PTR := POINTER(ARAY)
PTS := EBCDICARAY[5]
PINFO := PTR + 17
POUT := POINTER(INSTUFF[N],4)
```

Semantics

A *<pointer assignment>* is used to create a "pointer" which can then be used for various character purposes such as editing, testing, and scanning. (Refer to *<replace statement>* and *<scan statement>*.)

Pragmatics

A *<pointer assignment>* causes the creation of a "copy descriptor" of an array. The *<pointer variable>* (copy descriptor) can be set up with the needed *<character size>* via the *<pointer designator>* syntax.

CAUTION

Even though syntax allows it to be so, the *<pointer variable>* should not be global to the declaration of the array into which it "points." Total system failure can occur.

TASK ASSIGNMENT

Syntax

<task assignment> ::= *<task-valued task attribute>* := *<task designator>*
<task designator> ::= *<task identifier>* |
<task array identifier> [*<subscript list>*] |
MYSELF |
<task designator> . *<task-valued task attribute>*
<task-valued task attribute> ::= *<task designator>* . *<task-valued task attribute name>*
<task-valued task attribute name> ::= **EXCEPTIONTASK** |
PARTNER |
<task-valued task attribute name> . *<task-valued task attribute name>*

Examples

```
TISKIT.EXCEPTIONTASK := TASKIT
TSK.EXCEPTIONTASK := TASKARAY[N]
TASKVARB.PARTNER := COHORT
MYSELF.PARTNER := COWORKERS[INDX]
MYSELF.PARTNER.EXCEPTIONTASK := MYSELF.PARTNER.PARTNER
```

Semantics

As can be seen in the syntax, a *<task assignment>* is used to assign either of the *<task-valued attribute name>s*, **EXCEPTIONTASK** and **PARTNER**.

Briefly stated, the **EXCEPTIONEVENT** of a program's **EXCEPTIONTASK** will be **CAUSEd** whenever that program's status changes; e.g., suspended, terminated.

The **PARTNER** task attribute is used in conjunction with the *<continue statement>*.

ATTACH STATEMENT

Syntax

<attach statement> ::= **ATTACH** *<interrupt identifier>* **TO** *<event designator>*
<event designator> ::= *<event identifier>* |
<event array identifier> [*<subscript list>*] |
<event-valued task attribute>
<event-valued task attribute> ::= *<task designator>*. *<event-valued task attribute name>*
<event-valued task attribute name> ::= **EXCEPTIONEVENT**

Examples

```
ATTACH THEPHONE TO THEBELL
ATTACH ANSWERHI TO MYSELF.EXCEPTIONEVENT
```

Semantics

The *<attach statement>* associates an interrupt with an event. The association is such that causing the event interrupts the main program and places the interrupt code into execution (providing the interrupt is enabled; refer to the *<enable statement>*).

Pragmatics

While different interrupts can be simultaneously attached to the same event, a particular interrupt can at any one time be attached to only a single event. For this reason, if, at attach time, it is found that the interrupt is already attached to an event, it is automatically detached from the old event and then attached to the new event. Any pending invocations of the interrupt are lost.

It is possible to attach an interrupt to an event that is declared in a different *<block>*, for example, attach a local interrupt to a formal event. This can lead to certain compile-time or run-time **UP LEVEL ATTACH** errors if it is found potentially possible for the *<block>* containing the event to be exited prior to exiting the *<block>* that contains the interrupt.

BREAKPOINT STATEMENT

Syntax

<breakpoint statement>::= **BREAKPOINT**

Example

ON ANY FAULT, BEGIN BREAKPOINT; GO TO L; END;

Semantics

The *<breakpoint statement>* allows a user to interactively examine values of variables during the execution of a program.

Pragmatics

The execution of the *<breakpoint statement>* is a direct call on the **BREAKPOINT** intrinsic. This type of call may be used anywhere in the code; it is especially useful in an *<on statement>* or a software interrupt.

(Refer to the **BREAKHOST** and **BREAKPOINT** compiler options, appendix D, in order to create the necessary environment for interactive debugging using the *<breakpoint statement>*.)

CALL STATEMENT

Syntax

```
<call statement> ::= CALL <procedure identifier> <actual parameter part> [ <task designator> ]
<actual parameter part> ::= <empty> |
                            ( <actual parameter list> )
<actual parameter list> ::= <actual parameter> |
                            <actual parameter list> <parameter delimiter> <actual parameter>

<actual parameter> ::= <expression> |
                       <array designator> |
                       <direct file identifier> |
                       <direct switch file identifier> |
                       <event designator> |
                       <event array identifier> |
                       <file designator> |
                       <switch file identifier> |
                       <format designator> |
                       <switch format identifier> |
                       <label identifier> |
                       <switch label identifier> |
                       <list designator> |
                       <switch list identifier> |
                       <picture identifier> |
                       <procedure identifier> |
                       <task designator> |
                       <task array identifier>
<parameter delimiter> ::= , |
                          )" <letter string> "(
<letter string> ::={ any character string not containing a quote }
```

Examples

CALL COROOTEEN (X,Y,7, X+Y+Z) [TSK]
CALL HOME (OLDVAL,NEWVAL,FUNC) [TSKALAY[INDX]]

Semantics

The *<call statement>* initiates a procedure as a "co-routine". Initiation consists of setting up a separate stack, transferring any parameters that are passed, (by name or by value) and beginning the execution of its statements. Processing of the initiator is suspended.

The specified procedure cannot be typed.

Every co-routine has a "partner" task to whom control can be passed via the *<continue statement>*. The **PARTNER** task is set by default to the initiator, but may be changed by use of the appropriate *<task-valued task attribute>*.

Local variables and call-by-value parameters retain their values as control is passed to/from the co-routine.

There is a "critical block" in the caller's stack which cannot be exited until the co-routine is terminated. An attempt by the caller to exit that *<block>* before the co-routine is terminated will cause the caller (and all offspring) to be terminated.

A co-routine is terminated by exiting its own outermost block or by executing the statement "*<task designator>* . **STATUS** := -1;".

The *<actual parameter part>* must agree with the *<formal parameter part>* of the callee, or a run-time error will occur.

The *<task designator>* associates a task with the co-routine at initiation such that the MCP will set up the co-routine according to certain constraints such as **COREESTIMATE**, **STACKSIZE**, **DECLAREDPRIORITY**, and so forth. Refer to *<arithmetic task attribute>* and *<Boolean task attribute>*.

Pragmatics

As stated earlier, the *<call statement>* causes the initiation and set up of a separate stack as a co-routine. Because of the overhead involved, a co-routine should be established once and then used via *<continue statement>*s. If a *<call statement>* is used as a *<procedure statement>*, overall system efficiency will be severely degraded.

CASE STATEMENT

Syntax

<*case statement*>::= **CASE** <*arithmetic expression*> **OF** <*case body*>
<*case body*>::= **BEGIN** <*statement list*> **END** |
BEGIN <*numbered statement list*> **END**
<*numbered statement list*>::=<*numbered statement group*> |
<*numbered statement group*> ; <*numbered statement list*>
<*numbered statement group*>::=<*number list*> <*statement list*>
<*number list*>::=<*unsigned integer*>: |

ELSE : |

<*unsigned integer*> : <*number list*> |

ELSE : <*number list*>

<*statement list*>::= <*statement*> |
<*statement list*> ; <*statement*>

Examples

```
CASE I OF
BEGIN
    J := 1;
    J := 20;
    BEGIN
        J := 3;
        K := 0;
    END;
    J := 4;
END;

CASE I OF
BEGIN
    1:2:5:7:    J := 3;
                Q := J-1;
    3:4:20:     J :=4

  ELSE: GOTOBADCASEVALUE;
    END;
```

Semantics

The <*case statement*> provides a convenient means of dynamically selecting one of many alternative statements for execution at a particular point in the processing of a program. There are two types of <*case body*>*s*: implicitly numbered statements and explicitly numbered statements. The code is selected differently for each type.

IMPLICITLY NUMBERED STATEMENTS

The <*statement*> to be executed is selected by first evaluating the <*arithmetic expression*> in the <*case head*>. If its value is not integral, it is integerized by rounding and then used as an index to the statements in the <*case body*>. The N statements in the <*case body*> are numbered 0 to N-1. The <*statement*> corresponding to the index value is the <*statement*> executed. If the index value is less than zero or greater than N, an **INVALID INDEX** interrupt is generated. Only one <*statement*> in the <*case body*> can be selected each time the <*case statement*> is executed; however, this <*statement*> can be a <*compound statement*>, <*block*>, another <*case statement*>, or a null statement. (A null statement is a <*dummy statement*> that occupies a position in a <*case statement*>.)

EXPLICITLY NUMBERED STATEMENTS

This form of the <*case statement*> requires that the user explicitly number the statement groups. The <*numbered statement group*>*s* must lie within the range of O to N. If the integerized value of the <*arithmetic expression*> is less than O or greater than N or the integerized value is not associated with some <*statement list*> an **INVALID INDEX** interrupt is generated. However, if an **ELSE**: has been specified in a <*number list*>, control is transferred to the <*statement list*> following the **ELSE**:. At the end of each <*numbered statement group*> a branch is generated to the <*statement*> following the <*case statement*>.

Pragmatics

A <*case statement*> cannot have more than 1024 <*numbered statement group*>*s*.

CAUSE STATEMENT

Syntax

<*cause statement*> ::= CAUSE (<*event designator*>)

Examples

```
CAUSE (EVNT)
CAUSE (EVNTARAY[INDX])
CAUSE (TSK.EXCEPTIONEVENT)
```

Semantics

The <*cause statement*> activates all tasks that are waiting on the event. Normally the <*cause statement*> also sets the event to the **HAPPENED** state. (Refer to the <*waitandreset statement*> for exceptions.)

If there is an enabled interrupt attached to the event, each cause of the event will result in one execution of the interrupt code.

Pragmatics

"Activating a task" does not guarantee that the task goes into immediate execution. Activating a task consists of delinking the task from an event queue (each event has its own queue), and linking that task in priority order into a system queue called the **READYQ**. The **READYQ** is a queue of all tasks that are capable of running. Tasks are taken out of the **READYQ** when either a processor is assigned to the task or the task must wait for something such as an **I/O** operation or an event to be caused. A task will only be placed into actual execution when it is the top item in the **READYQ** and a processor is available.

A **CAUSE** of a **HAPPENED** event is essentially a no-op; i.e., the system does not "remember" every cause unless an interrupt is attached to the event.

CAUSEANDRESET STATEMENT

Syntax

<causeandreset statement> ::= CAUSEANDRESET (*<event designator>*)

Examples

CAUSEANDRESET (EVNT)
CAUSEANDRESET (EVNTARAY[INDX])
CAUSEANDRESET (MYSELF.EXCEPTIONEVENT)

Semantics

The *<causeandreset statement>* activates all tasks waiting on the event. It varies from the *<cause statement>* in that the resultant state of the event is set to **NOT HAPPENED.**

Pragmatics

Refer to *<cause statement>*.

CHANGEFILE STATEMENT

Syntax

<changefile statement>::= **CHANGEFILE**,(*<directory element>*,*<directory element>*)
<directory element>::= *<pointer expression>* |
<array row> |
<directory string>

<directory string>::= "*<filetitle>*." |
"*<filetitle>* **ON** *<packname>*."

Example Program

Program changes A/B to C/D and then removes C/D.

```
BEGIN
     ARRAY A, B [0:44];
     BOOLEAN B;
     POINTER PA, PB;
     PA:= POINTER (A[O]);
     PB:= POINTER (B[O]);
     REPLACE PA BY 8 "A/B.";
     REPLACE PB BY 8 "C/D.";
     IF B:= CHANGEFILE (PA,PB) THEN ERROR;
     IF B:= REMOVEFILE (8"C/D.") THEN ERROR;
END.
```

Semantics

The *<changefile statement>* changes the names of directories and files without opening them. The second *<directory element>* designates the title to which the first title is to be changed. If the change is on a pack, the second title must be followed by "**ON** *<packname>*.". An error is returned if the first title includes a packname. The *<changefile statement>* returns a value of **TRUE** if an error occurred. The error numbers, stored in [39:20], defining the failure are as follows:

a. 10 - first filename in error.
b. 20 - second filename in error.
c. 30 - filename has not been changed.

(Refer to the *<removefile statement>*.)

CHECKPOINT STATEMENT

Syntax

<checkpoint statement>::= **CHECKPOINT** (*<device>*, *<disposition>*)
<device>::= **DISK** |
DISKPACK |
PACK |
<disposition> ::= **LOCK** |
PURGE

Example

BOOL := CHECKPOINT (DISK, PURGE)

Semantics

The checkpoint/rerun facility gives the programmer a tool to protect a program against the disruptive effects of unexpected interruptions in its execution by periodically invoking the "checkpoint" procedures. This procedure takes a complete snapshot of the job and stores it on disk. The job can then be restarted in case its subsequent normal operation is interrupted. If a halt/load or other system interruption occurs, the job is restarted either at the last "no task active" point or, if the operator permits, at the last checkpoint, whichever is more recent. Checkpoint information can also be retained after a successful run to permit restarting a job to correct a bad data situation.

The *<device>* options determine the media to be used for the checkpoint files.

The *<disposition>* option **PURGE** removes all files at successful termination of the job and protects the job against system failures. The **LOCK** option saves all files indefinitely and can be used to RESTART a job even if it has terminated normally.

Values returned by the *<checkpoint statement>* as a result of an attempted checkpoint are as follows:

[0:1]	=	exception bit
[10:10]	=	completion code (refer to checkpoint/restart messages)
[25:12]	=	checkpoint number
[46:1]	=	restart flag (1 = restart)

Pragmatics

<u>Files</u>

When a checkpoint is invoked, the following files are created:

a. The checkpoint file – CP/<JN>/<CPN>
where <JN> is a four digit job number and <CPN> is a three digit checkpoint number. If the PURGE option has been specified, the checkpoint number is always zero and each succeeding checkpoint with purge removes the previous one (within the job). If the LOCK option is used, the checkpoint number starts at a value of one for the first checkpoint and is incremented by one for

each succeeding checkpoint with lock in the job. If the two types are mixed within a job, the "locked" checkpoints use the ascending number and the "purged" checkpoints use zero, leaving O-n at the completion of the job.

b. Temporary files – CP/<JN>/F<FN>
where <FN> is a three digit file number which starts at one and is incremented by one for each temporary disk or system resource pack file.

c. Job files – CP/<JN>/JOBFILE
This file is created under the **LOCK** option only.

The **LOCK/PURGE** option also has an effect when the task terminates. If the task terminates abnormally and the last checkpoint has used the **PURGE** option, then the checkpoint file (# 0) is changed to have the next sequential checkpoint number and the jobfile is created if necessary. If the job terminates normally and only purge checkpoints have been taken, the CP/<JN> directory is removed.

Restarting

There are two ways a job may be restarted:

a. After a Halt/Load. The system will automatically attempt to restart any job that was active at the time of a halt/load. If a checkpoint had been invoked since the last "no active task" point, then the operator will be given an RSVP to determine whether the job should be restarted. He can respond OK (restart at the last checkpoint), DS (don't restart), or QT (don't restart but save the files for later restart if it was a checkpoint with purge).

b. <rerun> statement. A job may be restarted programmatically in the Workflow Language by use of a <rerun statement>.

RERUN <JN>/<CPN>

Example: RERUN 1234/2 restarts job 1234 at checkpoint 2.

Conditions that can inhibit a successful restart are as follows:

a. Usercode invalid

b. OLAYROW different value

c. Program recompiled since checkpoint

d. Different MCP level

e. Different intrinsics

Restrictions

There are restrictions on the use of system features when used in conjunction with the checkpoint/rerun facility. These restrictions which will inhibit a successful checkpoint are summarized as follows:

a. Direct I/O (direct arrays or direct files)

b. Data Comm I/O (open data comm files)

c. Open DMS sets

d. Interprogram communication (the task being checkpointed must have no children or siblings, and its parent must be the WFL job)

e. Paper tape I/O

f. SPO files

g. Duplicated files

h. Output directly to line printer or card punch (backup is acceptable)

i. Task running in swap space

j. Checkpoints may not be taken inside sort input or output procedures (sort provides its own restart capability)

k. Checkpoints may not be taken in a compile-and-go program, as this creats an IPC environment

Further Considerations

For jobs which take a large number of checkpoints with lock, the checkpoint number counts up to 999 and then recycles to 1 (leaving 0 undisturbed). When this happens, the checkpoints previously numbered 1, 2, etc. are lost as new ones using those numbers are created.

If a temporary disk file is open at checkpoint, it is locked under the CP directory. If it is subsequently locked by the program, the name is changed to the current file title. At restart the file is looked for only under the CP directory resulting in a NO FILE condition. To avoid this, all files which are to be eventually locked should be opened with file attribute PROTECTION set to SAVE. (To remove the file, it must be closed with purge.) True temporary files which are never locked do not have this problem. All data files must be on the same medium as they were at the checkpoint. They do not have to be on the same units or the same locations on disk or disk pack. They must necessarily retain the same characteristics (blocking, etc.). The checkpoint/rerun system makes no attempt to restore the content of a file to the state it was in at the time of the checkpoint. The file is merely repositioned. At this time, volume numbers are not verified.

As a result of the IPC restriction, CANDE (and currently RJE) may not be used to run a program with checkpoints. The checkpoints will not be taken.

If a rerun is initiated and the job number is in use by another job, a new job number is supplied and the CP/<JN> directory node is changed to reflect the new job number.

When a job is restarted at some checkpoint which was not the last, subsequent checkpoints taken from the restarted job continue in numerical sequence from the one used for the restart. Previous high-numbered checkpoints are lost.

Checkpoint/Restart Messages

The following are a list of messages that can appear as the result of a checkpoint/restart.

CHECKPOINT MESSAGES	COMPLETION VALUE
CHECKPOINT #nnn	0
INVALID AREA IN STACK	1
SYSTEM ERROR	2
BAD IPC ENVIRONMENT	3
NO USER DISK FOR CP FILE	4
IO ERROR DURING CHECKPOINT	5
# ROWS IN CP FILE > 1024	6
DIRECT FILE NOT ALLOWED	7
TOO MANY TEMPORARY DISK FILES	8
PAPER TAPE FILE NOT ALLOWED	9
DUPLICATED FILE NOT ALLOWED	10
CON FILE NOT ALLOWED	11
CARD PUNCH FILE NOT ALLOWED	12
OPEN REVERSED TAPE FILE NOT ALLOWED	13
DISKHEADER IN STACK	14
DMS AREA IN STACK	15
DIRECT ARRAY IN STACK	16
DIRECT DOPE VECTOR IN STACK	17
SUBSPACE IN STACK	18
STACKMARK	19
SORT AREA IN STACK	20
REMOTE FILE NOT ALLOWED	21
ILLEGAL CONSTRUCT	22
BDBASE ILLEGAL	23
TEMP FILE ON NAMED PACK	24

CHECKPOINT

Continued

RESTART MESSAGES

RESTART PENDING (RSVP)
MISSING CHECKPOINT FILE
IO ERROR DURING RESTART
USECODE NO LONGER VALID
OPERATOR DSED RESTART
OPERATOR QTED RESTART
MISSING CODE FILE
NOT ABLE TO RESTART
INVALID JOB FILE
ERR COPYING JOB FILE
RESTART AS CP/nnnn
MISSING JOB FILE
FILE POSITIONING ERROR
WRONG JOB FILE
WRONG CODE FILE
BAD CHECKPOINT FILE
BAD STACK NUMBER
WRONG MCP

CLOSE STATEMENT

Syntax

<close statement> ::= **CLOSE** (*<file designator>*) |
CLOSE (*<file designator>* , *<close option>*)
<close option> ::= * | **PURGE** | **REEL** | **CRUNCH**

Examples

```
CLOSE (FILEID)
CLOSE (FILEID, *)
CLOSE (FILEID, PURGE)
CLOSE (FILEID, REEL)
CLOSE (FILEID, CRUNCH)
```

Semantics

The *<close statement>* causes the referenced file to be closed.

With no *<close option>*, the following actions take place:

a. On a card output file, a card containing an ending label is punched. The file must be labeled.
b. On a line printer file, the printer is skipped to channel 1, an ending label is printed, and the printer is again skipped to channel 1. The file must be labeled.
c. On an unlabeled tape output file, a double tape mark is written after the last block on tape and the tape is rewound.
d. On a labeled tape output file, a tape mark is written after the last block on the tape; then an ending label is written followed by a double tape mark and the tape is rewound.
e. On a disk file, if the file is a temporary file, the disk space is returned to the system.
f. The **I/O** unit is released to the system.

For all types of files, the buffer areas are returned to the system.

<close option>

If the "*" symbol is used and the file is a tape file, the **I/O** unit remains under program control and the tape is not rewound. This construct is used to create multi-file reels.

When the "*" symbol is used on multi-file input tapes, and **LABELTYPE = STANDARD**, the following actions can take place: If the value of the attribute **DIRECTION** is **FORWARD**, the tape is positioned forward to a point just following the ending label of the file; if the value of the attribute **DIRECTION** is **REVERSE**, the tape is positioned to a point just in front of the beginning label for the file; if the end-of-file branch has been taken, no action is performed to position the file. On a single-file reel, the action taken is the same as for a multi-file reel. The next reference to this file must be read in the opposite direction from that of the prior read on the file; otherwise the program encounters end-of-file.

When the "*" symbol is used and **LABELTYPE** is not **STANDARD**, the tape is spaced over the tape mark (or read) or a tape mark is written on output going forward. The essential difference is that with **OMITTEDEOF**, labels are not spaced over; but, with **STANDARD**, labels are spaced over.

If the **PURGE** option is used, the file is closed, purged, and released to the system. If the file is a permanent disk file, it is removed from the disk directory and the disk space is returned to the system.

If the **REEL** option is used, the file must be a multi-reel tape file. The current reel is closed and a subsequent reference of the file implicitly opens the next reel. This is provided primarily for the use of direct tape files, where the system does not automatically perform reel switching.

If the **CRUNCH** option is specified, the file must be a disk file. The unused portion of the last row (beyond the end-of-file indicator) of disk space is returned to the system. Note that the file cannot be "expanded", but can be written inside of the end-of-file limit.

NOTE

All combinations of the *<close statement>* which are not valid for the type of unit which is assigned to the file are equivalent to the *<rewind statement>*.

CONDITIONAL STATEMENT

Syntax

<conditional statement> ::= *<if statement>* |
<if statement> **ELSE** *<statement>* |
<iteration statement>

Examples

IF BOOL THEN GO TO EOJ
IF Q> VAL THEN X := O ELSE X := * + 1
IF NOGO THEN BEGIN . . . END ELSE BEGIN . . . END
WHILE BOOL DO . . .

Semantics

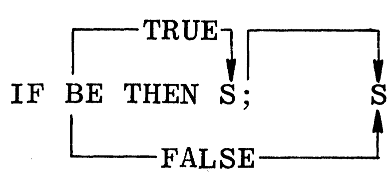
The *<conditional statement>* causes its constituent statements to be executed or skipped depending upon the logical value of the *<Boolean expression>*.

<if clause> *<statement>*

One of the permissible forms of a *<conditional statement>* is *<if clause>* *<statement>*. This form operates as follows: The *<statement>* following the sequential operator **THEN** is executed if the logical value of the preceding *<Boolean expression>* is **TRUE**; otherwise, that *<statement>* is ignored.

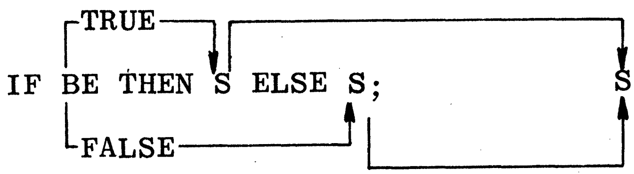
NOTE

In the examples that follow, **BE** represents any *<Boolean expression>*, and **S** represents any *<statement>*.



<if clause> *<statement>* **ELSE** *<statement>*

A second form of the *<conditional statement>* contains the sequential operator **ELSE**. This form of the *<conditional statement>* operates as follows: If the logical value produced by the *<Boolean expression>* is **TRUE**, the statement following the sequential operator **THEN** is executed and the statement following the sequential operator **ELSE** is ignored. If the logical value of the *<Boolean expression>* is **FALSE**, the statement following the sequential operator **ELSE** is executed and the statement following the sequential operator **THEN** is ignored.



NESTED *<conditional statement>s*

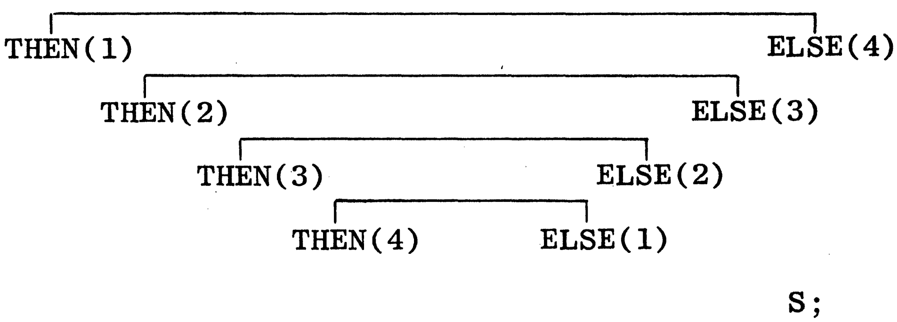
The statements following the delimiters **THEN** and **ELSE**, or both, may be conditional statements or a series of nested conditional statements.

The *<Boolean expression>s* in the *<if clause>s* of these statements are evaluated left-to-right in a manner similar to the evaluation of the conditional arithmetic expression.

When using nested *<conditional statement>s,* the programmer must remain aware of the necessity of maintaining correspondence between the delimiters **THEN** and **ELSE.**

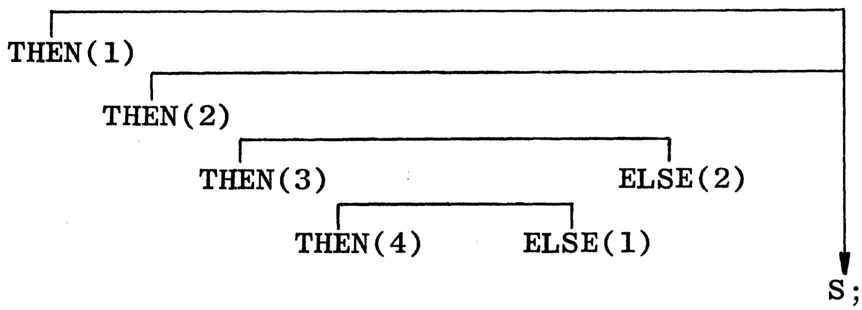
For explanatory purposes, assume that a given statement has equally matched **THEN-ELSE** pairs. In such a case, the innermost **THEN** and the immediately following (the innermost) **ELSE** are treated as one of pair, and from this center the pairs proceed outwards. This case is illustrated as follows:

Conditional S:



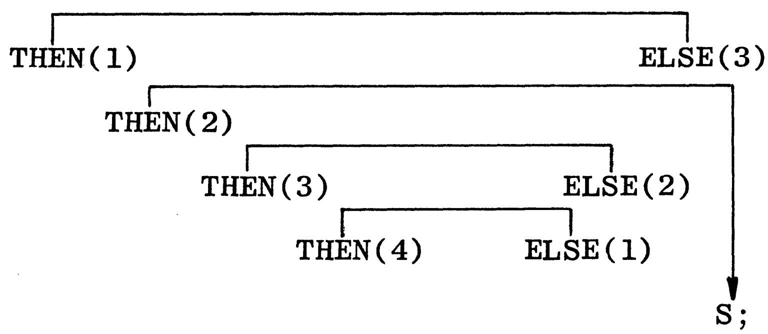
If **THEN** appears more often than **ELSE** in the statement, the pairs of delimiters are matched as described in the example above, and the first, and any following **THEN** not having a corresponding **ELSE**, causes the program to transfer to the next statement if the *<Boolean expression>* yields a value of **FALSE.** This case is illustrated as follows:

Conditional S:



In the case illustrated by:

Conditional S:



the **ALGOL** compiler would not produce the required result because **ELSE(3)** would be matched with **THEN(2)** and, if the *<Boolean expression>* preceding **THEN(1)** yielded a value of **FALSE**, the program would skip **ELSE(3)** and continue in sequence.

However, since a statement within a statement could itself be a *<compound statement>* or a *<block>*, the correspondence of the delimiters could be established clearly by defining the nested *<conditional statement>s* as *<compound statement>s*, the bracket words **BEGIN** and **END** indicating the different levels of nomenclature.

ENTERING A *<conditional statement>*

A *<go to statement>* may lead to a *<labeled statement>* within a *<conditional statement>*. The subsequent action is equivalent to what would occur if the *<conditional statement>* is entered at its beginning and evaluation of the *<Boolean expression>* causes the *<labeled statement>* to be executed.

CONTINUE STATEMENT

Syntax

<continue statement> ::= CONTINUE |
CONTINUE (*<task designator>*)

Examples

CONTINUE
CONTINUE (PROC1)

Semantics

The execution of *<continue statement>s* causes programmatic control to pass back and forth between co-routines.

Because the execution of *<continue statement>s* causes control to alternate between primary and secondary co-routines, processing always continues at the point where it terminated.

The secondary co-routine uses the form without the *<task designator>* to pass control back to its primary (often referred to as an "empty *<continue statement>*").

Pragmatics

A co-routine is a program (separate stack) that is established by means of a *<call statement>*. The "caller" is referred to as the "primary" and the "callee" as the "secondary".

DEALLOCATE STATEMENT

Syntax

<deallocate statement> ::= **DEALLOCATE** (*<array row>*)

Examples

```
DEALLOCATE (ARAY)
DEALLOCATE (MATRIXARY [INDX,*] )
```

Semantics

The *<deallocate statement>* causes the contents of the specified *<array row>* to be discarded and the memory area to be returned to the system.

Pragmatics

When the *<array row>* is deallocated, it is made not present (all data is lost). When the *<array row>* is used again, it is made present and each element is reinitialized to zero.

DETACH STATEMENT

Syntax

<detach statement> ::= **DETACH** *<interrupt indentifier>*

Examples

DETACH THEPHONE

Semantics

The *<detach statement>* severs the association of an interrupt to an event. Any pending invocations of the interrupt are discarded. Detaching an interrupt that is not attached is essentially a "no-operation", that is, no error mechanism invocation occurs.

The enabled/disabled condition of the interrupt is not changed by a *<detach statement>*; upon a subsequent *<attach statement>* the condition is the same as it was before the *<detach statement>*.

DISABLE STATEMENT

Syntax

<disable statement> ::= DISABLE |
DISABLE *<interrupt identifier>*

Examples

DISABLE
DISABLE THEPHONE

Semantics

The *<disable statement>* consisting simply of "**DISABLE**" (i.e., does not specify a particular *<interrupt identifier>*) is referred to as a "general disable" and, as such, a flag is set which causes the system not to look for interrupt code to execute for this task. The effect of this is as if all interrupts for the task have been set to their disabled state. During this period, all interrupts whose associated events are caused are placed in an interrupt queue of the task.

If the *<disable statement>* specifies an *<interrupt identifier>*, just that interrupt is disabled and the system queues them until subsequently enabled.

The purpose of queueing interrupts is to guarantee that no interrupts will be "lost" during the time they are attached. (See the *<attach statement>*.) Queueing continues until the appropriate *<enable statement>* is executed.

Note that disablement or enablement of an interrupt is independent of its being attached or detached to an event.

DISPLAY STATEMENT

Syntax

<display statement> ::= **DISPLAY** (*<pointer expression>*)

Examples

DISPLAY (POINTER (Q,8))
DISPLAY (PTR)

Semantics

The *<display statement>* causes the **EBCDIC** characters pointed at by the *<pointer expression>* to be displayed on the display console. The maximum number of characters allowed is 430. A message of less than 25 characters must be terminated by the character 4"00".

DO STATEMENT

Syntax

<do statement> ::= **DO** *<statement>* **UNTIL** *<Boolean expression>*

Examples

```
DO UNTIL FALSE
DO . . . UNTIL X=10000
DO BEGIN
     . . . ;
     . . . ;
     END
     UNTIL ALLDONE
DO J := J/2 UNTIL BUF[I:=I–J] LSS JOB
```

Semantics

The iterative *<do statement>* is executed as follows:

The *<do statement>* causes the *<statement>* following **DO** to be executed and then the *<Boolean expression>* to be evaluated. If the result is **FALSE**, the *<statement>* is executed again and the *<Boolean expression>* re-evaluated. This sequence of operations continues until the value of the *<Boolean expression>* is **TRUE** or until a *<go to statement>* is executed, at which time control passes to the specified destination or the next program *<statement>*. Figure 5–1 illustrates the **DO-UNTIL** loop.

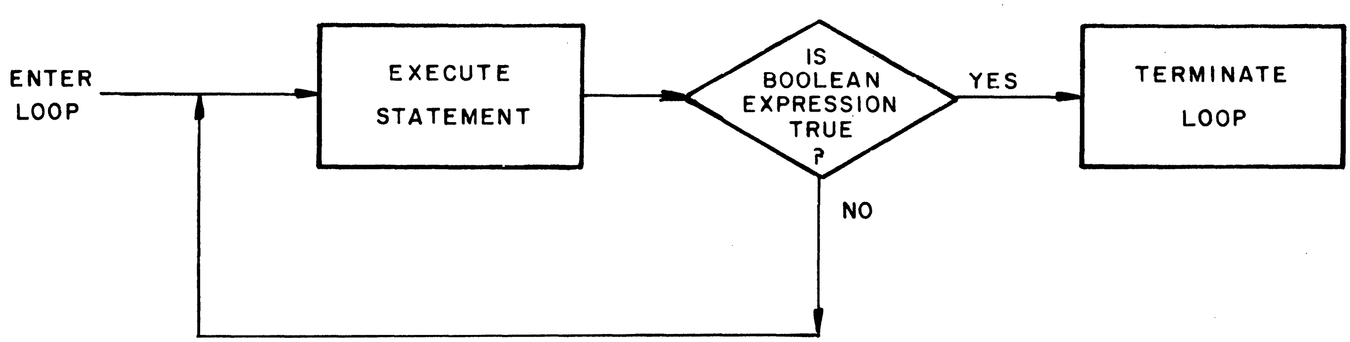


Figure 5–1. **DO-UNTIL** Loop

ENABLE STATEMENT

Syntax

<enable statement> ::= ENABLE |
ENABLE *<interrupt identifier>*

Examples

ENABLE
ENABLE THEPHONE

Semantics

An *<enable statement>* consisting simply of "ENABLE" (i.e., does not specify a particular *<interrupt identifier>*) is referred to as a "general enable" and, as such, the system is allowed to look for and place into execution all enabled interrupts that are in the interrupt queue of this task.

Previously disabled interrupts can be enabled by the task while in a general disabled state. These interrupts will be executed when the flag is reset by the general *<enable statement>*, if the associated event is caused after the interrupts have been enabled.

If the *<enable statement>* specifies an *<interrupt identifier>*, just that interrupt is enabled.

Note the enablement or disablement of an interrupt is independent of its being attached or detached from an event.

EVENT STATEMENT

Syntax

<event statement> ::= *<cause statement>* |
<causeandreset statement> |
<fix statement> |
<free statement> |
<liberate statement> |
<procure statement> |
<reset statement> |
<set statement> |
<wait statement> |
<waitandreset statement>

Examples

```
CAUSE (...
CAUSEANDRESET (...
FIX (...
FREE (...
LIBERATE ( ...
PROCURE (...
RESET (...
SET (...
WAIT ...
WAITANDRESET (...
```

Semantics

Events have two basic characteristics called **HAPPENED** and **AVAILABLE**. Each characteristic can be in either of two states: **TRUE** or **FALSE**, often referred to as the **HAPPENED** bit or the **AVAILABLE** bit. These characteristics can be interrogated via the **HAPPENED** and **AVAILABLE** Boolean intrinsics and can be changed via the *<event statement>s.*

EXCHANGE STATEMENT

Syntax

<exchange statement> ::= **EXCHANGE** (*<file designator>* [*<row/copy numbers>*] ,
<file designator> [*<row/copy numbers>*])

Examples

```
EXCHANGE (FILE1[ROW6], FYLE2[ROW0])
EXCHANGE (MASTERFYLE[ROW2, COPY3], REBUILTFYLE[ROW0, COPY1])
```

Semantics

The *<exchange statement>* can be used to "trade" rows between two disk files. The *<row number>* begins with zero (0), and the *<copy number>* begins with one (1). If a *<copy number>* is specified, then only the rows for that copy are exchanged.

Pragmatics

The referenced file(s) cannot be in an opened state when the *<exchange statement>* is executed. The two rows must be the same size. The specified *<row number>* as well as the specified *<copy number>* must be legitimate. The referenced disk file cannot be a "code file" of any kind.

If the system detects any type of error, the exchange is not actually performed and the program resumes its execution. After using the *<exchange statement>* the row addresses should be checked (via *<arithmetic file attribute>s*) to ensure that the exchange was successfully completed.

FILL STATEMENT

Syntax

<fill statement> ::= **FILL** *<array row>* **WITH** *<value list>*
<value list> ::= *<initial value>* |
<value list> , *<initial value>*
<initial value> ::= *<number>* |
<string> |
<unsigned integer> (*<value list>*)

Examples

```
FILL MATRIX[*] WITH 458.54, +546, – 1354.54@6, 16@–12
FILL GROUP[1,*] WITH .25, “ALGOL”, “ “ “, 4“FFFFF”, “365”
```

Semantics

The *<fill statement>* fills an *<array row>* with specified values. An initial value of the form *<unsigned integer>* (*<value list>*) uses the *<unsigned integer>* as a repeat count and repeats the *<value list>* the number of times indicated.

The row designator of the *<array row>* part indicates which row is to be filled, by designating a specific value for each subscript position of the *<array row>*. The symbol * must appear in the rightmost subscript position of the row designator. If the value of a row designator is other than integer, it is rounded to an integer in accordance with the rules applicable to *<assignment statement>s.*

Pragmatics

If the *<value list>* contains more values than the size of the *<array row>*, filling stops when the *<array row>* is full.

The comma in the *<value list>* causes word alignment of the next *<initial value>*.*<initial value>s* less than 48 bits are right-justified with leading zeros inserted in the word. *<initial value>s* greater than 48 bits are left-justified with trailing zeros inserted in the word.

If the size of array is longer than the supplied *<value list>*, the remainder of the *<array row>* is left “as is.”

The length of the *<value list>* cannot exceed 4095 48-bit words.

The *<fill statement>* cannot be used with character arrays.

FIX STATEMENT

Syntax

<fix statement> ::= **FIX** (*<event designator>*)

Examples

```
FIX (EVNT)
FIX (EVENTARAY [INDX])
IF GOTIT := FIX (FYLELOCK) THEN
FIX (MYSELF.EXCEPTIONEVENT)
```

Semantics

Upon completion of the execution of the *<fix statement>*, the *<event designator>* referenced is **NOT AVAILABLE.**

The *<fix statement>* (conditional procure function) is a Boolean function that examines the available state of an event. If the state is **AVAILABLE**, the event is procured, the state set to **NOT AVAILABLE**, and a **FALSE** returned. If the available state is **NOT AVAILABLE**, the function returns a **TRUE**, leaving the available state unchanged.

FOR STATEMENT

Syntax

<for statement> ::= **FOR** *<variable>* := *<for list>* **DO** *<statement>*
<for list> ::= *<for list element>* |
<for list> , *<for list element>*
<for list element> ::= *<initial part>* *<iteration part>*
<initial part> ::= *<arithmetic expression>*
<iteration part> ::= *<empty>* |
STEP *<arithmetic expression>* **UNTIL** *<arithmetic expression>* |
STEP *<arithmetic expression>* **WHILE** *<Boolean expression>* |
WHILE *<Boolean expression>*

Examples

```
FOR I := 0 DO ...
FOR J := 1 STEP 1 UNTIL 255 DO . . .
FOR INDX := 0, 1, 2, 10, 15, 37, 5, 16 DO ...
FOR X := 0 STEP 1 UNTIL 5, 29, 47 STEP 3 UNTIL LIM DO ...
FOR NXT := BEG STEP AMT WHILE NOT DONE DO ...
FOR N := IX + 7 WHILE TARGET LEQ RANGE DO ...
```

Semantics

The *<variable>* in the *<for statement>* is referred to as the controlled variable. The *<for statement>* can be best understood by isolating the following three distinct operational steps:

a. Value assignment to the controlled variable.
b. Test of the limiting condition.
c. Execution of the *<statement>* following **DO.**

Each type of *<for list element>* specifies a different process. However, all have one property in common, which is, the initial value assigned to the controlled variable is that of the leftmost *<arithmetic expression>* in the *<for list element>s.*

The *<for list element>* determines what values are to be assigned to the controlled variable and what test to make of the controlled variable to determine whether or not the *<statement>* following **DO** is executed. When a *<for list element>* is exhausted, the next element in the *<for list>* is considered, progressing from left-to-right. When all *<for list element>s* have been utilized, the *<for list>* is considered exhausted and control continues with the next *<statement>* following the *<for statement>*. It is possible for the *<statement>* following **DO** to transfer control outside the *<for statement>*, in which case some *<for list element>s* may not be exhausted when the *<for statement>* is exited.

<arithmetic expression> <empty>

The format for this variation is as follows:

FOR V : = AEXP1, AEXP2,. . . , AEXPN DO S_{do}**; S**

When the <*for list element*> is simply an <*arithmetic expression*>, there is only one value to be assigned to the controlled variable. Since there is no limiting condition, no test is made. After assignment of the <*initial part*> to the controlled variable, the <*statement*> following **DO** is executed. If more than one <*initial part*>, the <*initial part*>*s* are assigned to the controlled variable consecutively until the <*for list*> is exhausted.

Figure 5–2 illustrates the **FOR-DO** loop.

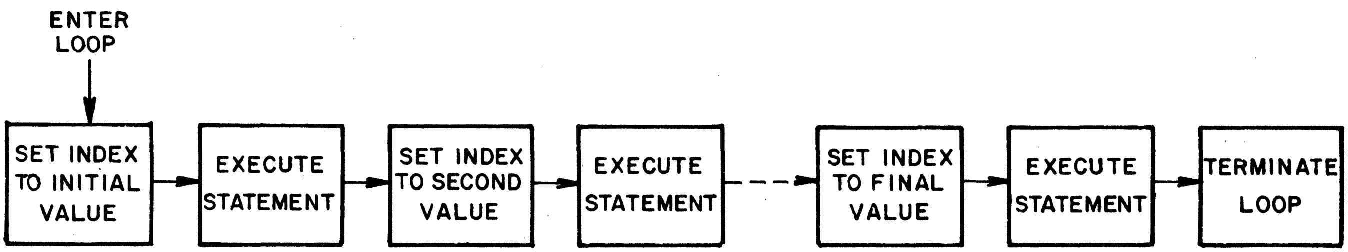


Figure 5–2. **FOR-DO** Loop

STEP <*arithmetic expression*> **UNTIL** <*arithmetic expression*>

When the <*for list element*> is of the form:

FOR V:= AEXP1 STEP AEXP2 UNTIL AEXP3

where **AEXP1, AEXP2,** and **AEXP3** represent <*arithmetic expression*>*s,* the process described below is used. A new value is assigned to the controlled variable, V, each time the <*statement*> following **DO** is executed. First, an initial value, that of **AEXP1**, is assigned to the controlled variable. All subsequent assignments are equivalent to: **V := V + AEXP2,** and made immediately after the <*statement*> following **DO** is executed. The limiting condition on the value of V is given by **AEXP3,** which is evaluated anew each time through the loop.

A test is made immediately after each assignment of a value to V (including the first) to determine whether whether or not the value of V has passed **AEXP3.** Whether **AEXP3** is an upper or lower limit depends on the sign of **AEXP2. AEXP3** is an upper limit if **AEXP2** is positive, and a lower limit if **AEXP2** is negative. If **AEXP3** is an upper limit, then V has "passed" **AEXP3** when the expression **V LEQ AEXP3** is no longer **TRUE.** If **AEXP3** is a lower limit, then **V** has "passed" **AEXP3** when the expression **V GEQ AEXP3** is no longer **TRUE.** If **V** has not passed **AEXP3**, the <*statement*> following **DO** is executed. Otherwise, the <*for list element*> is exhausted. Figure 5-3 illustrates the **FOR-STEP-UNTIL** loop.

Note

A step of O is not allowed in the <*for statement*>.
A run-time error will occur.

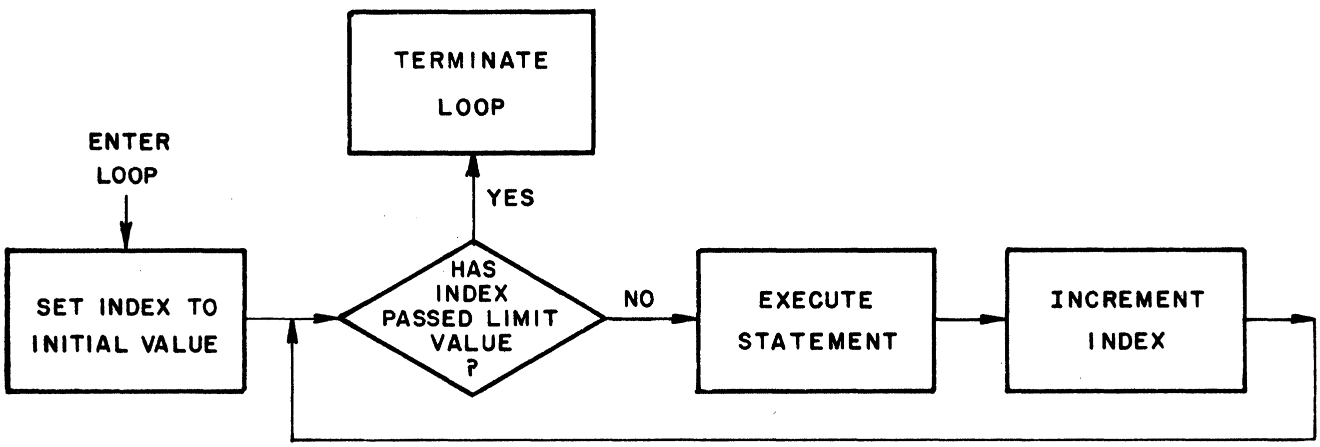


Figure 5–3, **FOR-STEP-UNTIL** Loop

STEP <*arithmetic expression*> **WHILE** <*Boolean expression*>

When the <*for list element*> is of the form

FOR V := AEXP1 STEP AEXP2 WHILE BEXP DO S_{do}; **S**

where **AEXP1** and **AEXP2** are <*arithmetic expression*>*s* and **BEXP** is a <*Boolean expression*>, the process is described below. A new value is assigned to the controlled variable V if the **BXEP** is **TRUE** each time the statement following **DO** is executed. First, the initial value **AEXP1** is assigned to the controlled variable. All subsequent assignments are equivalent to **V := V+AEXP2** and are made immediately after the <*statement*> following **DO** is executed. After each assignment to **V**, the <*Boolean expression*> **BEXP** is evaluated, and as long as **BEXP** is **TRUE**, the <*statement*> following **DO** is executed. This can be stated concisely as follows:

V := AEXP1

L3: IF BEXP THEN BEGIN S_{do}; **V := V+AEXP2; GO TO L3 END;**
S

Figure 5–4 illustrates the **FOR-STEP-WHILE** loop.

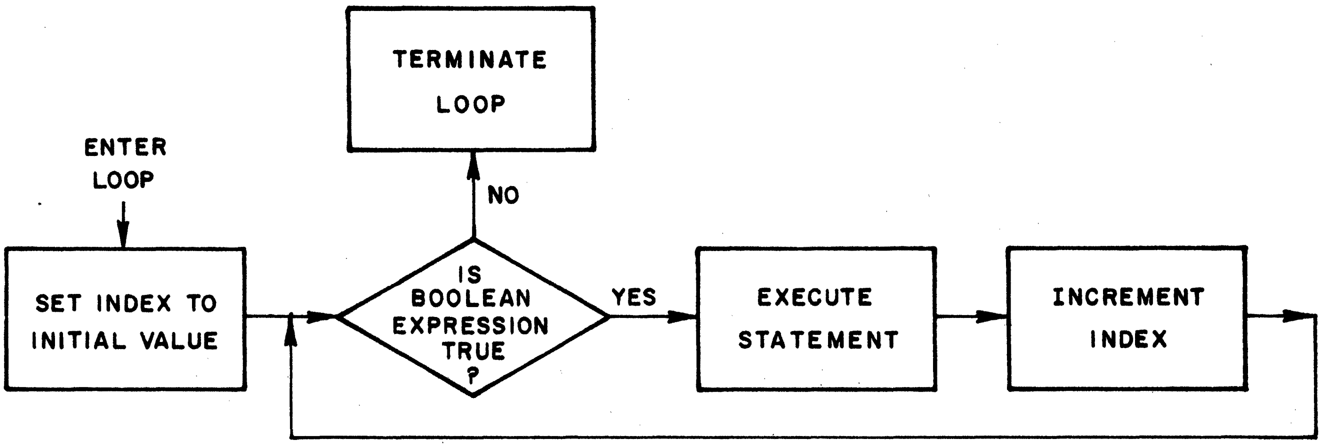


Figure 5–4. **FOR-STEP-WHILE** Loop

WHILE *<Boolean expression>*

When the *<for list element>* is of the form **WHILE** *<Boolean expression>*, the controlled variable is assigned the value of the *<initial part>*. A test is then made of the *<Boolean expression>* following **WHILE**. If the logical value is **TRUE**, the *<statement>* following **DO** is executed. This process continues, with the *<for list element>s* being assigned to the controlled variable, until the value of the *<Boolean expression>* is **FALSE**, at which time control is transferred to the next *<statement>* in the *<program>*. For example,

FOR V := V + 1 WHILE V LEQ 5 DO . . .

If V had the value of zero before execution of this statement, the statement between the **BEGIN-END** delimiters would have been executed five times.

Figure 5–5 illustrates the **FOR-WHILE** Loop.

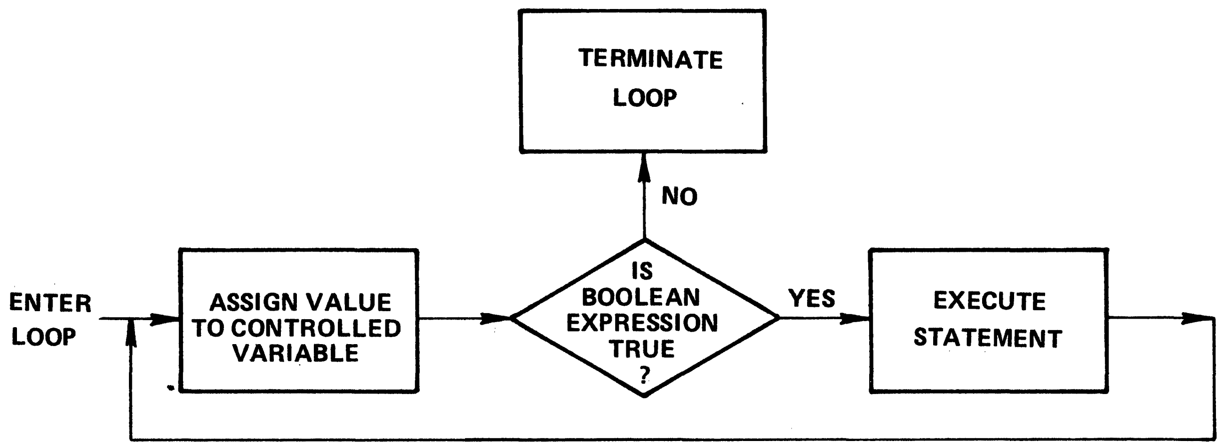


Figure 5–5. **FOR-WHILE** Loop

FREE STATEMENT

Syntax

<free statement> ::= FREE (*<event designator>*)

Examples

FREE (EVNT)
FREE (EVNTARAY [INDX])
IF WASPROCEDURED := FREE (FYLELOCK) THEN . . .

Semantics

This statement, of arbitrary value and dangerous use, unconditionally resets the event state to **AVAILABLE.** It does not activate any task suspended by an attempt to procure the event nor does it activate any task waiting on the event.

The *<free statement>* can be used as Boolean function that returns a **FALSE** if the event is already **AVAILABLE,** and a **TRUE** is returned if the event was **NOT AVAILABLE.** In either case, the event is unconditionally reset to **AVAILABLE.**

GO TO STATEMENT

Syntax

<go to statement> ::= **GO TO** *<designational expression>* |
GO *<designational expression>*

Examples

```
GO TO LABEL1
GO LABEL2
GO TO IF K=1 THEN SELECT [2] ELSE START
GO TO SELECTIT [INDX]
```

Semantics

The *<go to statement>* transfers control to the *<label>* that is the value of the *<designational expression>*.

If the *<designational expression>* specifies an invalid designation (only possible when using a *<switch label declaration>*), control passes to the *<statement>* following the *<go to statement>*.

Labels must be declared in, and therefore are local to, the innermost block in which they appear as a statement label. A *<go to statement>* cannot lead from outside a *<block>* to a point inside that *<block>*; each *<block>* must be entered at the *<block head>* so that the associated declarations can be invoked.

NOTE

Refer to *<designational expression>* for an explanation of the "bad go to."

I/O STATEMENT

Syntax

<*I/O statement*> ::= <*accept statement*> |
<*close statement*> |
<*display statement*> |
<*lock statement*> |
<*read statement*> |
<*rewind statement*> |
<*seek statement*> |
<*space statement*> |
<*write statement*>

Semantics

An <**I/O** *statement*> causes information to be exchanged between a program and its peripheral device(s), or it allows the programmer to perform certain control functions.

The <*accept statement*> and <*display statement*> are unique in that the programmer is not required to specify a "file" to/from which data is transferred. These two statements have a very limited syntax and therefore are completely described elsewhere in this manual.

The remaining <**I/O** *statement*>*s* all reference a file which must be declared by the programmer (refer to <*file declaration*>)

A full treatment of **ALGOL I/O** is beyond the scope of this manual, but it is necessary to point out that there are two distinct methods of **I/O** which the programmer can do. The first and typical method is referred to as "normal **I/O**" and the second method is called "Direct **I/O**". These two methods are explained separately under each of the <**I/O** *statement*>*s* which can be found elsewhere in this manual. Briefly, however, the major difference between Normal **I/O** and Direct **I/O** has to do with "buffering", or the overlap of program execution and **I/O** operations. Whereas Normal **I/O** is normally overlapped (i.e., it is automatic), Direct **I/O** can be used to achieve or avoid overlap as desired.

NORMAL I/O

Unless a file is declared to be **DIRECT**, it is by default handled in Normal **I/O** fashion. The amount of buffering between the <**I/O** *statement*>*s* and program execution depends on the number of buffers allocated for the file (refer to <*file declaration*>).

A Normal **I/O** <*read statement*> causes the automatic testing of the availability of the needed record. The program is suspended in the <*read statement*> until such time as the record is actually available for use.

A <*write statement*> in Normal **I/O** transfers the specified data to a buffer and the program is immediately released to begin execution of the next <*statement*>. If all the buffers are full at the instant of the <*write statement*>, the program is suspended until such time as a buffer is available.

DIRECT I/O

Direct **I/O** brings the programmer closer to the actual input/output operations. In certain situations, it may be necessary to avoid any suspension of the program for any reason whatsoever. In other situations, it may be necessary to perform non-standard **I/O** operations as well as to mask certain types of error conditions which could arise.

To perform Direct **I/O** on a file (call it **FID**) the file must be declared as a **DIRECT** file. (Refer to *<file declaration>*.)

The syntax for Direct **I/O** read or write operation employs the *<arithmetic expression>*, *<array row>* form of *<format and list part>*. Optional *<action labels>* of the [*<event designator>*] can be used. The *<array row>* is called the user's **I/O** area, and the *<direct array identifier>* must be used for the *<array name>* part in the *<array row>* construct. Thus to Direct read 10 words from **FID** into Direct array **A** using the event **EVT**, **READ(FID,10,A[*]) [EVT]** would be written. When executing this *<statement>*, the **MCP** establishes a relationship between the **I/O** area and the event **EVT**.

However, before any subsequent use of the **I/O** area can be made in the program, either for calculations or for further **I/O**, the Direct **I/O** operation must be finished. The event mechanism can be used by having **EVT** caused when the read operation is finished. The event can be inspected by means of the Boolean intrinsic function **HAPPENED**, or by obtaining the **I/O** result descriptor, through the use of the **WAIT** intrinsic on the Direct array. The user can also use a *<wait statement>* on the event to de-activate the process until the event is caused. Once the operation has been completed, the event should be reset before reusing it (see the *<waitandreset statement>*).

In Direct **I/O**, the **I/O** operations analogous to **SPACE** and **REWIND** are performed as if they are a read or write operation, except that the **IOCW** direct array attribute is specifically set to the proper hardware **IOCW** for the operation.

IF STATEMENT

Syntax

<if statement> ::= *<if clause>* *<statement>*
<if clause> ::= IF *<Boolean expression>* THEN

Examples

```
IF ALLDONE THEN GO AWAY
IF ENDITALL := X=0 THEN
   WHILE A > COWNT DO . . .
```

Semantics

The *<if statement>* provides a means of making a conditional transfer of control based on data or results of a computation.

The *<statement>* following **THEN** is executed if the *<Boolean expression>* results in a **TRUE** condition. (Refer to *<conditional statement>* for more information on the use of the *<if statement>*.)

INTERRUPT STATEMENT

Syntax

<interrupt statement> ::= *<attach statement>* |
<detach statement> |
<disable statement> |
<enable statement>

Examples

```
ATTACH ...
DETACH ...
DISABLE ...
ENABLE ...
```

Semantics

A process can be interrupted upon the occurrence of a specific event if an interrupt has been declared, attached to the event, and enabled. The paragraphs that follow describe briefly the *<interrupt statement>*. (Refer to the specific *<interrupt statement>* for more detail.)

<attach statement> and *<detach statement>*

The *<attach statement>* is used to associate an interrupt with an *<event designator>*. The *<attach statement>* does not implicitly enable or disable an interrupt. If it has not been disabled, it is enabled.

The *<detach statement>* is used to sever the association between the interrupt and the event to which it has been attached.

<enable statement> and *<disable statement>*

The *<enable statement>* and *<disable statement>* are used to explicitly enable and disable an interrupt if one is specified. If none is specified, then all interrupts are enabled or disabled.

INVOCATION STATEMENT

Syntax

<invocation statement> ::= *<call statement>* |
<procedure statement> |
<process statement> |
<run statement>

Examples

CALL . . .
PROCEDURE . . .
PROCESS . . :
RUN . . .

Semantics

An *<invocation statement>* causes a previously declared procedure to be executed as a subroutine, an asynchronous process, a co-routine, or an independent program.

With the exception of the *<procedure statement>*:

a. A separate stack is always initiated, and
b. The specified procedure cannot be typed.

With the exception of the *<run statement>*, parameters may be passed by name or value. All parameters passed in the *<run statement>* must be by value.

ITERATION STATEMENT

Syntax

<iteration statement> ::= *<do statement>* |
<for statement> |
<thru statement> |
<while statement>

Examples

```
DO BEGIN. . . ; END UNTIL SWEAT
FOR X:= 0 STEP 1 UNTIL 5,29,47 STEP 3 UNTIL LIMIT DO . . .
THRU MAXI := REAL (PTR,3) DO . . .
WHILE INDX LEQ MAXVAL DO . . .
```

Semantics

<iteration statement>s provide methods of forming loops in a *<program>*. They allow for repetitive execution of a *<statement>* zero or more times.

The various iterative mechanisms are described as follows:

a. The *<do statement>* causes the statement following **DO** to be executed and then the *<Boolean expression>* to be evaluated. If the result is **FALSE** the *<statement>* is executed again; if **TRUE**, control passes outside the *<do statement>*.
b. The *<for statement>* assigns an initial value to a controlled variable. It then proceeds to execute and increment that variable until the limit has been passed.
c. The *<thru statement>* tests a repeat index, executes a *<statement>*, and then decrements the repeat index by one.
d. The *<while statement>* evaluates a *<Boolean expression>*, and if **TRUE**, the statement is executed. If **FALSE**, control is passed outside the *<while statement>*.

LIBERATE STATEMENT

Syntax

<liberate statement> ::= **LIBERATE** (*<event designator>*)

Examples

LIBERATE (EVNT)
LIBERATE (EVNTARAY[INDX])

Semantics

The *<liberate statement>*, when executed, produces several effects. First, the procure list is examined. If there are no other tasks waiting to procure the event, the event state is set to **AVAILABLE.** If there are other tasks waiting to procure the event, the event state is left marked as **NOT AVAILABLE.** Also, all tasks waiting on the event are activated, that is, an implicit cause is executed. This can result in a change to the **HAPPENED** state of the event, depending on whether the tasks that are waiting have used *<wait statement>* or the *<waitandreset statement>*.

Pragmatics

Even though all waiting tasks are activated, they are linked into the **READYQ** in priority order (see the *<cause statement>*). At this point, all tasks will attempt to procure the event (see the *<procure statement>*).

LOCK STATEMENT

Syntax

<lock statement> ::= **LOCK** (*<file designator>* *<lock option>*)
<lock option> ::= *<empty>* |
, **CRUNCH** |
, *

Examples

```
LOCK (FILEA)
LOCK (FYLE, CRUNCH)
LOCK (FYLE, *)
```

Semantics

The *<lock statement>* causes the referenced file to be closed. If the file is tape, it is rewound and a system message is printed that notifies the operator that the reel must be removed and saved. If the file is not a disk file, the unit is made inaccessible to the system until the operator readies it again manually. If the file is a disk file, it is retained as a permanent file on disk. The file buffer areas are returned to the system.

A *<lock statement>* which has a non-empty *<lock option>* performs the same action as the *<close statement>* which specifies **CRUNCH.** The file must be a disk file. The unused portion of the last row (beyond the end-of-file indicator) of disk space is returned to the system. The disk file can no longer be expanded without being copied into a new file; however, the file can be written inside of the end-of-file limit.

MERGE STATEMENT

Syntax

<merge statement> ::= MERGE (*<output option>* ,
<compare procedure> ,
<record length> ,
<merge option list>)
<merge option list> ::= *<merge option>* |
<merge option list> , *<merge option>*
<merge option> ::= *<input option>*

Example

MERGE (LINEOUT, COMP, 14, IN1, IN2)

Semantics

The *<merge statement>* causes data in all of the files specified by the *<merge option list>* to be combined and returned. The *<compare procedure>* determines the manner in which the data is combined. The *<output option>* specifies how the data is to be returned from the merge.

The *<merge option list>* must contain between two and eight input options, inclusive which must be files or Boolean procedures.

The *<output option>*, *<compare procedure>*, *<record length>*, and *<input option>* are as specified for the *<sort statement>*.

For more detailed information concerning the *<merge statement>*, refer to the B 6000 Series Operation Guide Reference Manual.

MULTIPLE ATTRIBUTE ASSIGNMENT STATEMENT

Syntax

<multiple attribute assignment statement> ::= *<file identifier>* (*<initial attribute list>*)

Example

FYLE (BUFFERS = 3, INTMODE = 3, KIND = DISK)
LINE(TITLE = P, INTNAME = Q);

Semantics

One intrinsic call **SET**s all attributes except when a *<pointer-valued attribute name>* appears in the *<initial attribute list>* and is initialized to a *<pointer expression>*, rather than a string. In this case, the compiler generates an intrinsic call distinct from the call that **SET**s the rest of the attributes in the list.

A Boolean file attribute followed by a comma is assigned a value of **TRUE**; that is, in a *<file declaration>*, **OPTIONAL** has the same effect as **OPTIONAL = TRUE**.

ON STATEMENT

Syntax

```
<on statement> ::= <enabling on statement> |
                   <disabling on statement>
<enabling on statement> ::= ON <fault list> <fault information part> , <fault action> |
                            ON <fault list> <fault information part> : <fault action>
<fault list> ::= <fault name> |
                 <fault list> OR <fault name>
<fault name> ::= ANYFAULT | EXPONENTOVERFLOW |
                 EXPONENTUNDERFLOW | INTEGEROVERFLOW | INVALIDADDRESS |
                 INVALIDINDEX | INVALIDOP | INVALIDPROGRAMWORD |
                 LOOP | MEMORYPARITY | MEMORYPROTECT |
                 PROGRAMMEDOPERATOR | SCANPARITY |
                 STRINGPROTECT | ZERODIVIDE
<fault information part> ::= <empty> |
                             [ <fault stack history> ] |
                             [ <fault stack history> : <fault number> ] |
                             [ : <fault number> ]
<fault stack history> ::= <array row> |
                          <pointer expression>
<fault number> ::= <variable>
<fault action> ::= <statement>
<disabling on statement> ::= ON <fault list>
```

Examples

```
ON ZERODIVIDE OR INVALIDINDEX [:A[J]],
    BEGIN
    END
ON ANYFAULT;
ON MEMORY PROTECT OR LOOP : Q := 2;
ON ZERODIVIDE OR INTEGEROVERFLOW;
ON ANYFAULT [POINTR + 2 : Z], HANDLFALTS(Z);
ON EXPONENTOVERFLOW[A[*]], RECOVER(A);
ON ANYFAULT [:J]:
    BEGIN
    END;
```

Semantics

The *<on statement>* is used to enable or disable an interrupt for one of 15 fault conditions. The interrupt remains enabled until one of the following occurs:

a. The procedure or *<block>* that contains the *<on statement>* is exited.
b. The interrupt is explicitly disabled.
c. A new interrupt is enabled for the same fault condition.

Whenever the *<block>* that contains an *<on statement>* is exited, the interrupt status for that fault condition reverts to whatever was enabled when the *<block>* was entered.

The interrupt *<statement>* must either be an unconditional **GO TO**, a *<compound statement>*, or a *<block>* that contains an unconditional **GO TO**, since the interrupted code cannot be resumed.

The *<fault list>* enables the user to arm several faults with respect to the same *<fault action>* (refer to the first and third examples, above), or to disarm one or more faults at the same time (refer to the fourth example, above). Note that the occurrence of any one of the faults in the *<fault list>* is sufficient to cause transfer of control to the *<fault action>*.

The non-*<empty>* *<fault information part>* provides the user with the stack history at the time of the occurrence of the fault, and/or the number corresponding to the fault kind (useful only when more than one fault is armed with respect to the same *<fault action>*). The *<fault number>*, when indicated, is set to one of the following values upon occurrence of the corresponding fault:

VALUE	FAULT
1	ZERODIVIDE
2	EXPONENTOVERFLOW
3	EXPONENTUNDERFLOW
4	INVALIDINDEX
5	INTEGEROVERFLOW
7	MEMORYPROTECT
8	INVALIDOP
9	LOOP
10	MEMORYPARITY
11	SCANPARTIY
12	INVALIDADDRESS
14	STRINGPROTECT
15	PROGRAMMEDOPERATOR
18	INVALIDPROGRAMWORD

The format of the **STACKHISTORY** is the standard format:

SSS:AAAA:Y,#SSS:AAAA:Y,#...,#SSS:AAAA:Y.

or

SSS:AAAA:Y#(DDDDDDDD),# . . . ,#SSS:AAAA:Y#(DDDDDDDD).

where **SSS** is the segment number, **AAAA** is the address, **Y** is the syllable, # is a blank space, **DDDDDDDD** is the line number (only present if **LINEINFO** has been **SET** during program compilation). The period (.) always terminates the last entry.

Thus, in the fifth example, above, the **STACKHISTORY** begins at **POINTER + 2** and continues until either the area or the **STACKHISTORY** information is exhausted.

Note that the *<fault stack history>* and the *<fault number>* are fixed, with respect to address, when the *<on statement>* is executed; that is, when the fault is armed, not when the fault occurs. Thus, in the *<on statement>*

```
ON ZERODIVIDE [A[I,*]:B[J]] : Q:=B[J] + Q
```

The *<array row>* **A[I,*]** is determined by the value of **I** at the execution of the *<on statement>*, and not when any **ZERODIVIDE** actually occurs. This is also true for the *<variable>s* **B[J]** and **J.**

The form **ON** *<fault list>* *<fault information part>* : *<fault action>*, which is known as the "go to" form, does not require the user to do a "bad go to" out of the *<fault action>*; the bad go to is performed by the system. Consequently, program control can continue from the *<fault action>*. For example,

```
    BEGIN
    ARRAY Z, Q [0:9999];
      .
      .
      .
    READ (FIL, 10000, Q);
    I:=0;
    ON ZERODIVIDE: Q[I]: = 1.0E-47;
L:Z[I]:=SQRT(6.32/Q[I]);
    IF (I:=I + 1) < 10000 THEN GO TO L1;
      .
      .
      .
    END;
```

This example uses the hardware to check the value of **Q [I]** for zero, instead of doing so explicitly (the former is generally faster).

Note that the "environment" (i.e., stack) for the "go to" case has been cut back before control is transferred to the *<fault action>*.

When using the form **ON** *<fault list>* *<fault information part>*, *<fault action>*, the user cannot do a "go to" to a label inside the *<fault action>* from outside the *<fault action>*.

Unexpected results occur when a "go to" to a formal label (label passed as a parameter) is attempted.

The *<disabling on statement>* disables, or disarms, those faults corresponding to the *<fault name>s* in the *<fault list>*.

No block exit is required to deactivate the armed faults for the block.

PROCEDURE STATEMENT

Syntax

<procedure statement> ::= *<procedure identifier>* *<actual parameter part>*

Examples

```
SIMPL
HEAVY (X,Y,A [*], SQRT (BINGO) + BASE)
FANCY (P1) "IS THE FIRST, THE NEXT IS SECOND" (P2) "THEN THIRD" (P3)
```

Semantics

A *<procedure statement>* causes a previously declared procedure to be executed.

A typed procedure returns a value. However, when a typed procedure is used as a *<procedure statement>*, this value is ignored.

The *<actual parameter list>* of the *<procedure statement>* must have the same number of entries as the *<formal parameter list>* of the *<procedure declaration>* heading.

Formal and actual parameters must correspond in "type" and "kind" of quantities. The correspondence is obtained by taking the entries of these two lists in the same order. Parameters may be passed by name or by value.

The normal use of a *<procedure statement>* is such that when the procedure is entered, program control is suspended at the point of the invocation until the referenced procedure "falls out" the end. At that time, program control resumes at the *<statement>* following the invocation. However, this would not be the case if within the referenced procedure, a "bad go to" is executed. (Refer to *<procedure declaration>*, *<go to statement>*, and *<designational expression>*.)

PROCESS STATEMENT

Syntax

<process statement> ::= **PROCESS** *<procedure identifier>* *<actual parameter part>* [*<task designator>*]

Examples

```
PROCESS AGENT [TSK]
PROCESS ACHILD (OURARRAY,YOUREVENT[INDX],COWNT) [TSKARAY[INDX]]
```

Semantics

The *<process statement>* initiates a procedure as an asynchronous process. Initiation consists of setting up a separate stack for the process, transferring any parameters which are passed (by name or by value) and beginning the execution of its statements. The initiator then resumes execution and both are run in parallel (or concurrently, depending on processor availability).

The specified procedure cannot be typed.

An asynchronous process depends upon the initiator's stack for globals or call-by-name parameters. Thus for each process there is, in the initiator's stack, a "critical block" which cannot be exited until the process is terminated. The "critical block" is the *<block>* of highest lexicographic level which contains the declaration of the procedure itself, call-by-name parameters, or the *<task designator>*. This may be the *<block>* containing the *<process statement>*, the outer *<block>* of the program, or some *<block>* in between. An attempt by the initiator to exit that *<block>* before the process is terminated will cause the initiator and all its offspring to be terminated.

A process can be terminated by exiting its own *<block>* or by executing the statement "*<task designator>*.**STATUS** := -1;".

The *<actual parameter part>* must agree with the *<formal parameter part>* of the process, or a run-time error will occur.

The *<task designator>* associates a task with the process at initiation such the MCP will set up the process according to certain constraints such as **COREESTIMATE, STACKSIZE, DECLAREDPRIORITY,** and so forth. Furthermore, various task attributes can be interrogated while the process is running. (Refer to *<arithmetic task attribute>* and *<Boolean task attribute>*.)

PROCURE STATEMENT

Syntax

<procure statement> ::= **PROCURE** (*<event designator>*)

Examples

PROCURE (EVNT)
PROCURE (EVNTARAY[INDX])

Semantics

The *<procure statement>* tests the available state of an event. If the event is **NOT AVAILABLE**, the task is suspended until some other task executes the *<liberate statement>*. If the event was **AVAILABLE**, the event state is set to **NOT AVAILABLE**, and the task continues in sequence.

Pragmatics

The *<procure statement>* provides a convenient method of sharing various resources by different programs/tasks. A convention should be established that a certain multi-usable resource or resources should not be used until a program/task has procured an event which is defined as the interlock. When its program/task has completed its use of the resource(s), it should execute a *<liberate statement>* on the event. (Refer to the *<liberate statement>*.)

PROGRAMDUMP STATEMENT

Syntax

<programdump statement> ::= **PROGRAMDUMP** *<optional parameters>*
<optional parameters> ::= *<empty>* |
(*<parameter list>*)
<parameter list> ::= *<parameter item>* |
<parameter list> , *<parameter item>*
<parameter item> ::= **ARRAY** | **ARRAYS** | **BASE** | **CODE** | **DBS** | **FILE** |
FILES | **ALL** | *<arithmetic expression>*

Examples

PROGRAMDUMP
PROGRAMDUMP (ARRAYS)
PROGRAMDUMP (ARRAYS, BASE, CODE, FILE)
PROGRAMDUMP (ALL)
PROGRAMDUMP (DUMPPARAM)
PROGRAMDUMP (ARRAYS, DBS)

Semantics

Execution of the *<programdump statement>* causes the **MCP** to print out (using the program's **TASKFILE**) the stack of the program. Several options are available as to which items of the stack are to be dumped and analyzed.

If the *<optional parameters>* has the form of *<empty>*, the stack is printed/analyzed according to the specifications in the program's **OPTION** word. (See Task Attributes.)

If the contents of the program's arrays are to be printed, the option **ARRAYS** must be specified.

The bottom (or "base") of the user's stack will be printed if the **BASE** option is specified. The **MCP** uses the same portion of each stack to contain various words needed to control, identify, and log the program.

The **DBS** *<parameter item>* causes the output of data base stacks and structure information blocks. **DBS** turns on bit 15 of the option word.

Pragmatics

A programmer can explicitly **WRITE** his own diagnostic/debugging information to the **TASKFILE** such that the **PROGRAMDUMP** and his information are coordinated (refer to *<write statement>*).

The Segment Dictionary of the program is printed out as a separate stack if the **CODE** option is specified. Furthermore, the actual code will be printed for only those segments which are referenced by the program at the time of the *<programdump statement>*. Note that **VALUE ARRAY**s in the Segment Dictionary will be printed when both **CODE** and **ARRAYS** are specified.

If a program wants its files to be printed/analyzed, the **FILES** option must be specified. As each file is encountered, each word of the **FIB** is separately named and, in some cases, analyzed. Various bits of the *<arithmetic expression>* indicate the desired items:

7:1 = 1	If the **BASE** of the user stack is to be dumped.
8:1 = 1	If all encountered **ARRAYS** are to be printed.
9:1 = 1	If **CODE** (i.e., the Segment Dictionary stack) segments are to be printed.
10:1 = 1	If **FILES** are to be printed/analyzed.
10:4 = 15	If **ALL** portions of the program are to be printed/analyzed.
15:1 = 1	If the **DBS** data base stacks and structures are to be printed.

When the MCP has completed printing/analyzing the specified items, control passes to the next executable statement.

READ STATEMENT

Syntax

<read statement> :: = **READ** (<file part> <format and list parts>)
<action labels or finished event>

<file part >::=<file designator> <record number or carriage control> |
<core-to-core part>

<record number or carriage control> ::= <empty> |
[<arithmetic expression>] |
[**LINE** <arithmetic expression>] |
[**NO**] |
[**SKIP** <arithmetic expression>] |
[**SPACE** <arithmetic expression>] |
[**STACKER** <arithmetic expression>] |
[**STATION** <arithmetic expression>] |
[**STOP**] |
[**TIMELIMIT** <arithmetic expression>]

<core-to-core part> :: =<core-to-core file part> <core-to-core blocking part >
<core-to-core file part> :: = <array row> |
<pointer expression> |
<subscripted variable>

<core-to-core blocking part> :: = <empty> |
(<core-to-core record size>) |
(<core-to-core record size>,
<core-to-core records per file part>)
<core-to-core record size> :: =<arithmetic expression>

<core-to-core records per file part> ::= <arithmetic expression>

<format and list part> :: = <empty> |
, <format designator> | , <format designator>, <list> |
, ⟨<editing specifications>⟩ |
, ⟨<editing specifications>⟩, <list> |
, * , <list> | , <free field part> , <list> |
, <arithmetic expression> , <array row> |
, <arithmetic expression> , <subscripted variable> |
, <arithmetic expression> , <pointer expression> |

<list> ::= <list identifier> | <list segment> | <switch list identifier>
[<subscript>]

<free field part> ::= <asterisk part> <number of columns>
<slash part> <column width>

<asterisk part> ::= <empty> | *

<number of columns> ::= *<empty>* | [*<arithmetic expression>*]

<slash part> ::=/ | / /

<column width> ::= *<empty>* | [*<arithmetic expression>*]

<array row> ::= *<array name>* | *<array name>* [*<row designator>*]

<action labels or finished event> ::= *<emtpy>* |
[*<label 1>* : *<label 2>* : *<label 3>*] |
[*<label 1>* : *<label 2>*] |
[*<label 1>* : *<label 3>*] |
[*<label 1>*] |
[: *<label 2>* : *<label 3>*] |
[: *<label 2>*] | [: *<label 3>*] |
[*<event designator>*]

<label 1> ::= *<designational expression>*

<label 2> ::= *<designational expression>*

<label 3> ::= *<designational expression>*

NOTE

On any formatted I/O statement (excluding core-to-core I/O), the number of characters allowed in the I/O record is determined solely by the **MAXRECSIZE** of the file. If the format requires more characters than contained by the record to satisfy the list, a format error will result at run-time.

Examples

READ (*<file part>* *<format and list part>*)

READ (FILEID)
READ (FILEID, FMT)
READ (FILEID, FMT, LISTID)
READ (FILEID, *, LISTID)
READ (SPOFILE, FMT, A,B,C,)
READ (SPOFILE, /, SIZE, LENGTH, MASS)
READ (FILEID, FMT, 7, 2, A, B, C, ARRAY [A] , B+C.F)
READ (FILEID, /, J, FOR I := 0 STEP 1 UNTIL J DO ARRAY [I])

```
READ (FILEID,*,A,B,C,FOR A :=B*A STEP C UNTIL J DO ARRAY [I])
READ (SWFILEID[IF X > N THEN X+N ELSE O], 25, ARRY[2.*])
READ (FILEID, / , SWLISTID[I])
READ (FILEID, FMT, SWLISTID[I])
READ (SPOFILE, SWFMT[16] , A,B,C)
```

READ (*<file part>* *<format and list part>*) *<action labels or finished event>*

```
READ (FILEID) [EOFL:PARL]
READ (FILED, / , L,M,N,ARRAY[2]) [EOFL]
READ (FILEID[3] [NO]) [:PARL]
READ (SWFILEID[14] [NO] , A+EXP(B),ARRY[I,J,*]) [:PARSWL[M]]
READ (FILEID [NO] , SWFMT[6+J] , LISTID) [EOFSWL [Q*3]]
READ (SWFILEID[A+B] , * , SWLISTID[2+H/K]) [EOFL:PARL]
READ (FILEID[NO]) [EOFSWL[I] :PARSWL[J]]
READ (FYLE) [EOFL:PARL:DATAERRL]
READ (DIRFYLE) [EVNT]
READ (DIRFYLE, 30, DIRARAY) [EVNT]
```

Semantics

The *<read statement>* allows data to be assigned to various program variables. The result of this *<statement>* depends on the form of the *<file part>* element and on the form of the *<format and list part>* element.

NOTE

Because the syntax of the *<read statement>* and the *<write statement>* are identical, the pragmatic differences between the syntactical items are explained in the folowing paragraphs.

<file part>

READ

The *<file part>* form indicates where the data is to be found.

WRITE

The *<file part>* indicates where the data is to be written. WRITE (MYSELF.TASKFILE: . .) allows the user to write to the program's taskfile (refer to *<programdump statement>*).

<record number or carriage control>

READ

If the *<record number or carriage control>* element is *<empty>*, the record addressed by the pointer is read; the record pointer is adjusted to point to the next record in the file.

If the *<record number or carriage control>* element is an [*<arithmetic expression>*], its value indicates the relative address of the record in a file that is to be read. The record pointer is set to the specified address before the read is performed; the record pointer is not adjusted after the READ operation.

If the *<record number or carriage control>* element is [**NO**], the buffer is not released after it has been read or written; i.e., the record can be read again, perhaps with a different format.

If the *<record number or carriage control>* element is [**SPACE** *<arithmetic expression>*] the number of records specified in the *<arithmetic expression>* is skipped. Spacing is forward if the *<arithmetic expression>* is positive; backwards if negative.

If the *<record number or carriage control>* element is [STATION *<arithmetic expression>*], the last station attribute is set to the value of the *<arithmetic expression>*.

The [**TIMELIMIT** *<arithmetic expression>*] (relevant for REMOTE files only) element is a positive real number in units of seconds (fractional amount is allowed). If **TIMELIMIT** is zero (0), an indefinite wait is initiated. When the **TIMELIMIT** is greater than zero and no input is received within **TIMELIMIT** seconds, the *<read statement>* is terminated with a **TIMELIMIT** error.

A **TIMELIMIT** error is reported by the logical I/O result descriptor having the attention bit [0:1] and bit [15:1] turned ON. For example, **IF IORSLT : = READ (RMT [TIMELIMIT 15], 12, A) THEN IF IORSLT.** [15:1] **THEN % TIMELIMIT EXCEEDED GO AWAY;**

Note that the **TIMELIMIT** attribute becomes **SET**. This will effect all read/write operations within the program.

WRITE

If the *<record number or carriage control>* part is a [**LINE** *<arithmetic expression>*] and the file is a line printer file, then the printer spaces forward to the specified line before printing. However, the following must be observed:

a. The **PAGESIZE** file attribute must be SET or declared to be the number of lines on a page.

b. Since normal default action for ALGOL is print-before-carriage-action, a subsequent *<write statement>* can overprint the line.

c. The line number is RESET when [SKIP 1] is used.

The [**SKIP** *<arithmetic expression>*] part causes the line printer to skip to the channel indicated by the *<arithmetic expression>* after printing the current record.

The [**SPACE** *<arithmetic expression>*] part causes the line printer to space the number of lines denoted by the *<arithmetic expression>* after printing the current record. On other types of devices it causes the number of records signified by the *<arithmetic expression>* to be spaced.

If the specified file is remote, the [**STOP**] part does not do a line feed or a carriage return.

If the file is not a printer file, the *<record number or carriage control>* part is interpreted as a record number as described previously under the *<read statement>*.

The [**STACKER** *<arithmetic expression>*] part allows pocket selection for card punch files. Legal values for the arithmetic expression are 0 or 1. A 0 selects the normal pocket ; 1 selects the alternate pocket.

The [**STATION** *<arithmetic expression>*] part sets the **LASTSTATION** attribute to the value of the *<arithmetic expression>*.

If, when using the [**TIMELIMIT** *<arithmetic expression>*] part, the buffer does not become available within **TIMELIMIT** seconds, the write operation is terminated with a **TIMELIMIT** error.

CORE-TO-CORE I/O

<core-to-core part>

The *<core-to-core part>* indicates internal data transfer (i.e., no physical device is involved). If the *<core-to-core blocking part>* is *<empty>*, correct action will be taken for the *<core-to-core file part>*, just as it would be for a normal I/O statement; however, core-to-core I/O will be much faster. If the *<core-to-core blocking part>* is non-*<empty>*, the size and number of records into which the *<core-to-core file part>* is to be blocked can be specified.

<core-to-core file part>

For **HEX**, **BCL** or **EBCDIC** array rows or pointers as the *<core-to-core file part>* the default record size (i.e., the number of characters considered to be in the record) is dependent upon the character size of the array row or pointer and is determined by the actual length of the designated string.

For single and double precision array rows or subscripted variables, the default record size is computed by multiplying the length of the array row (or remaining length of the array row when a subscripted variable is used) times the number of characters per word, where characters per word is derived from the following table:

		(default character size)	
		BCL	EBCDIC
(precision)	single	8	6
	double	16	12

<core-to-core blocking part>

To specify a record size smaller than the default size, a value may be provided for the *<core-to-core record size>*. This value will always refer to record size in terms of characters. By supplying a value for *<core-to-core records per file part>*, the file part may be blocked into more records than the default value of one.

With formatted I/O, if the format requires more records than indicated by the *<core-to-core records per file part>*, a run-time error will be given. Another consideration is that the format may require more characters than the *<core-to-core file part>* contains. This will also result in a run-time error. In such a case, the number of characters indicated in the *<core-to-core blocking part>* (this number is computed by multiplying *<core-to-core record size>* times *<core-to-core records per file part>*) may appear to be large enough to satisfy the format, but the *<core-to-core blocking part>* may indicate more characters than the *<core-to-core file part>* actually contains. The programmer must take care to ensure compatibility between the *<core-to-core file part>*, the *<core-to-core blocking part>* and the format to avoid run-time errors.

Examples

```
         .
         .
         .
         REAL B,C;
         ARRAY A[0:9];
         .
         .
         .
   EX1:  READ (A(80), <T50,A6,I10> ,B,C);
         .
         .
         .
   EX2:  WRITE(A(15,3), <X5,I15>,1,2,3);
         .
         .
         .
   EX3:  WRITE(A(20,2), <X5,I15> ,1,2,3);
         .
         .
         B:="bbITEM";
   EX4:  WRITE(A(15,4),<".",X2,A6,I2,X4>,B,1,B,2,B,3,B,4);
         .
         .
         .
```

The statement labeled EX1 would result in a run-time error (FORMAT ERROR #217) because the format requires 65 characters, but the file part (array A) contains only 60 characters.

The statement labeled EX2 would result in a run-time error (FORMAT ERROR #117) because the format requires 20-character records, but 15-character records were specified in the blocking part.

The statement labeled EX3 would result in a run-time error (FORMAT ERROR #120) because the 3 list elements will require repeating the format 3 times. Thus, 3 records are required but only 2 records were specified in the blocking part.

The statement labeled EX4 would fill array A with the following **EBCDIC** data:

.bbbbITEMb1bbbb.bbbbITEMb2bbbb.bbbbITEMb3bbbb.bbbbITEMb4bbbb

<*format and list part*>

Read

The <*format and list part*> element indicates the program variables to which file data is to be assigned and the manner in which the data is to be interpreted in assigning it to these variables.

If the <*format and list part*> element is <*empty*> the input record is skipped.

A <*format designator*> without a <*list*> indicates that the referenced format contains a <*string*> into which corresponding characters of the input data are to be placed. The <*string*> in the format declaration is replaced by the <*string*> in the input data.

A <*format designator*> with a <*list*> indicates that the input data is to be edited according to the specifications of the referenced <*format declaration*> and assigned to the variables of the <*list*>.

The symbol *, together with a <*list*>, specifies that the input data is to be processed as full words, and that it is to be assigned to the variables of the <*list*> without being edited. The number of words read is determined by the number of <*variables*>in the<*list*> or the maximum record size, whichever is smaller.

An <*arithmetic expression*> followed by an <*array row*>, <*subscripted variable* > element or <*pointer expression*> specifies that input data is to be processed as full words, and that is to be assigned, without being edited, to the elements of the designed <*array row*>, <*subscripted variable*> element, or the item referenced by the <*pointer expression*>. The maximum record size, the number of elements in the <*array row*>. <*subscripted variable*> element or the item referenced by the <*pointer expression*>, or the value of the <*arithmetic expression*> determines the number of words read, depending upon which is the smallest. If Direct I/O is not being used, and the UNITS attribute=1, and INTMODE≠0, then all counts represent characters, not words.

FREE-FIELD I/O

The use of a free-field designator with the READ or WRITE statements allows I/O to be performed with editing, but without using a format statement. The appropriate format is selected automatically, but variations of the free-field designator give the user some control over the form of the output.

The general form for a free-format designator is:

ar/sw

where a is an optional asterisk (*), s is an optional second slash (/), and r and w are optional single precision arithmetic expressions enclosed in brackets.

Input

On input, only the simplest form consisting of a single slash (/) can be used. It allows input from records consisting of data items separated by commas.

All blanks are ignored. Character strings must be enclosed by quote marks (").

The symbol /, together with a *<list>* specifies that the input data is represented in a free-field format. All free-field input is in the form of *<free-field data>*.

The "syntax" for *<free-field data>* is as follows:

<free-field data> ::= *<field>* .*<field delimiter>* |
<free-field data> *<field>* *<field delimiter>*
<field> ::= *<empty>* | *<number>* | *<string>* |

<field delimiter> ::= . |
<letter> { any proper string not containing a comma}. |
{if the field is a /. the end of the current record serves as a field delimiter }

Examples

1,
2.5,
2.48 @ -20,
2 @ 34,
"THIS IS A STRING",
1 DELIMITER,
2.5 ANY COMMENT OR NOTE NOT CONTAINING A COMMA,
2.48 @ -20 VALUE FOR Z* (-3),
2 @ 34 ET CETERA,

Each field, except the slash (/), is associated with the list element to which it corresponds according to position.

All blanks in *<free-field data>* except those in strings are completely ignored.

Fields are handled as follows:

a. A number that is represented as an integer is converted as type **INTEGER** unless it is larger than the largest allowable integer, in which case it is converted as type **REAL**. Numbers that contain a decimal fraction are converted as type **REAL**.

b; Strings can be of any length. Each list element receives six or eight characters, depending on character size, until either the list or the string is exhausted. If the number of characters in the string is not a multiple of six, the last list element receives the remaining characters of the string. The string characters are stored right-justified in the list elements.

c. An *<emtpy>* field causes the corresponding list element to be ignored.

d. The / field causes the remainder of the current buffer to be ignored. The buffer following the slash is considered the beginning of a new field: therefore, the slash field does not require, or recognize, any field delimiter other than the end of the buffer in which it occurs. A slash field has no effect on list elements. The slash is a field by itself and must not be placed within another field or between a field and its delimiter.

e. The asterisk (*) field terminates the *<read statement>*. The program continues with the next statement in sequence. The list element corresponding to the asterisk remains unchanged, as do any subsequent elements in the list.

The logical values, for the purpose of free-field input, are as follows: an integer 1 (one) must be used in lieu of the logical value TRUE, and an integer 0 (zero) must be used in lieu of the logical value FALSE.

Output

On output, each value is edited into an appropriate format. An edited item is never split across a record boundary. If the record is too short to hold any reasonable representation of the item, a string of pound signs (#) is output in place of the item.

Data items are normally separated by a comma and a space. If the optional second slash (/ /) is used, they are separated by two spaces. Note that output produced in this manner cannot be read by a free-field input statement.

If the optional asterisk is used, the name of the data item and an equal sign (=) are output prior to the value of the data item. If the data item is not a variable name then the expression is output as the name of the data item.

It is not uncommon for users of free-field I/O to want to control spacing of items; hence this feature is now offered.

With columnized free-field output, each list element is output in a separate column. This process is controlled by two column factors. These factors are the number (r) of columns per record and the width (w) of each column, where w is measured in characters. Both r and w are integerized if necessary.

If r is zero, the number of columns per record will be determined from the value of w and the record length. If w is zero, the width of each column will be determined from the value of r and the record length. If both r and w are zero, there is no column structure to the output. If r and w are such that r columns of w characters cannot fit on one record, adjustments are made to both r and w. Note that the width of a column does not include the two-character delimiter; i.e., r*(w+2) must be less than or equal to the length of the record.

Example

```
ARRAY B [0:3];
WRITE (F, /,*A,*X+Y,*"HELLO",*7.2,
     *B [A],*SIN (X),*B,*PNTR FOR 3);
produces
A = 3.2, bX+Y = 2.4E+41, b HELLO, b CONST = 7.2, b
          B[3] = -82.173, b SIN(X) = 0.241392156792, b
          B[0] = 0.0, b B[1] = 0.0, b, B[2] = 682.173, b
          B[3] = -82.173,b PNTR=QZ#, b
```

Write

The *<format and list part>* part indicates which *<variable>s* contain the data and how the data is to be interpreted.

If the *<format and list part>* is *<empty>*, a blank record is written. A *<format identifier>* alone indicates that the referenced *<format declaration>* contains one or more strings that constitute the entire output.

A *<format identifier>* followed by a *<list>* indicates that the variables in the *<list>* are to be placed in a format, according to the specifications of the *<format declaration>*, and written as output.

The * symbol followed by a *<list>* or *<list identifier>* specifies that the variables in the *<list>* are to be processed as full words and are to be written by the number of variables in the *<list>*or the maximum block length, whichever is smaller. When unblocked records are used, the buffer size is the maximum record length.

An *<arithmetic expression>* used with an *<array row>* , *<subscripted variable>* , or *<pointer expression>* specifies that the elements of the designated *<array row>*,*<subscripted variable>* part, or item referenced by the *<pointer expression>* are to be processed as full words and are to be written as output without being edited. The number of words written is determined by the number of elements in the *<array row>*, *<subscripted variable>* part, or item referenced by the *<pointer expression>*, the maximum block length, or the absolute value of the arithmetic expression, whichever is smallest. When unblocked records are being used, the buffer size is the maximum record length. If the UNITS attribute = 1, the INTMODE ≠ 0, then all counts represent characters, not words.

<write statements> that do not reference a *<format declaration>* provide a faster output operation than those that require data to be edited.

<action labels or finished event>

<action labels or finished event> provide a means of transferring control from a *<read statement>*, *<write statement>*, or *<space statement>* when exception conditions occur. A branch to *<label 1>* takes place when an end-of-file condition occurs. A branch to *<label 2>* takes place if an irrecoverable parity error is encountered. A branch to *<label 3>* takes place if there is a conflict between the format and the data. If the appropriate label is not provided when an exception condition occurs, the program is terminated.

The [<*event designator*>] form can be used only for Direct I/O; the event is caused when the I/O operation is finished. (Refer to the DIRECT I/O paragraph.) <*action labels or finished event*> cannot be used with the following read/write construct: <*array row*>, <*arithmetic expression*>, <*array row*>.

Exception conditions occuring during a <*read statement*> or <*write statement*> can also be handled without the use of <*action labels or finished event*>. The I/O result word returned by the MCP I/O routines can be used as a Boolean primary. Refer to B 6000 Series Operation Guide Reference Manual, form 5001563, for a description of the contents of the I/O result word when an exception condition occurs.

For example,

```
IF BOOL := READ(FILEID, 14, A[*]) THEN GO TO ERRORCOND;
```

When exception conditions are handled in this manner, <*action labels or finished event*> cannot be used; the user assumes all responsibility for handling exception conditions. Furthermore, this method cannot be used for Direct I/O or <*read statements*> of the form: READ (<*array row*>, <*arithmetic expression*>, <*array row*>). Also, <*write statement*>*s* of the form: **WRITE** (<array row>, <*arithmetic expression*>. <*array row*>) are excluded.

A common I/O exception condition is break on output for remote programs. For example,

```
IF IORSLT := WRITE (RMT,12,A) THEN
    IF IORSLT.[13:1] THEN % BREAK ON OUTPUT
        BEGIN CLEANUP; GO XIT; END;
```

NOTE

Additional information pertaining to I/O operations can be found under the <*I/O statement*>.

REMOVEFILE STATEMENT

Syntax

<removefile statement> ::=**REMOVEFILE** (*<directory element>*)

Example

BOOL:=
REMOVEFILE ("MYTEST ON PACKFOUR.")

Semantics

The *<removefile statement>* provides the ability to remove directories and files without opening them. The *<removefile statement>* also returns a value of **TRUE** if an error occurred. The error numbers, stored in [39:20], defining the failure are as follows:

a. 10 - filename in error

b. 30 - filename not removed

If a *<pointer expression>* is used as a *<directory element>*, it must point to an array that contains the filetitle to be removed.

(Refer to the *<changefile statement>*.)

REPLACE STATEMENT

Syntax

```
<replace statement> ::= REPLACE <destination> BY <source list>
<destination> ::= <update pointer> <pointer expression>
<update pointer> ::= <empty> |
                     <pointer variable> :
<source list> ::= <source part> |
                  <source list> , <source part>
<source part> ::= <source> <transfer part> |
                  <arithmetic expression> <optional unit count> |
                  <digit convert part> |
                  <numeric convert part> |
                  <translate part> |
                  <pointer-valued attribute>
<source> ::= <update pointer> <pointer expression>
<transfer part> ::= <unit count> |
                    WITH <picture identifier> |
                    <scan part>
<unit count> ::= FOR <arithmetic expression> <units>
<units> ::= <empty> |
            WORDS
<scan part> ::= <condition> |
                FOR <count part> <condition>
<condition> ::= WHILE <relational operator> <arithmetic expression> |
                UNTIL <relational operator> <arithmetic expression> |
                WHILE IN <truthset table> |
                UNTIL IN <truthset table>
<count part> ::= <residual count> <arithmetic expression>
<residual count> ::= <empty> |
                     <simple variable> :
<truthset table> ::= <subscripted variable> |
                     <truthset identifier> |
                     ALPHA |
                     ALPHA6 |
                     ALPHA7 |
                     ALPHA8
<optional unit count> ::= <empty> |
                          <unit count>
<digit convert part> ::=
    <arithmetic expression> FOR <arithmetic expression> DIGITS |
    <arithmetic expression> FOR * DIGITS
<numeric convert part> ::=
    <arithmetic expression> FOR * NUMERIC |
    <arithmetic expression> FOR <count part> NUMERIC
```

<translate part> ::= *<source>* **FOR** *<arithmetic expression>* **WITH** *<translate table>*
<translate table> ::= *<subscripted variable>* |
<translatetable identifier> |
ASCIITOBCL |
ASCIITOEBCDIC |
ASCIITOHEX |
BCLTOASCII |
BCLTOEBCDIC |
BCLTOHEX |
EBCDICTOASCII |
EBCDICTOBCL |
EBCDICTOHEX |
HEXTOASCII |
HEXTOBCL |
HEXTOEBCDIC

Examples

```
REPLACE PTR BY "A"
REPLACE PTR:PTR BY "*" FOR 75
REPLACE PTR BY ITEM
REPLACE PRT BY (4"03").[7:48] FOR 1
REPLACE PTR BY " " FOR N WORDS
REPLACE PTR:PTR BY PST FOR 18
REPLACE PTR BY PST:PST FOR NUM WORDS
REPLACE PTR BY PINFO WITH PIC
REPLACE PTR:PTR BY PST WHILE NEQ " "
REPLACE PTR BY PST WHILE IN ALPHA
REPLACE P BY X FOR * DIGITS
REPLACE P BY X FOR 50 NUMERIC
REPLACE P BY X FOR * NUMERIC
REPLACE PTR BY PST WHILE IN MYTRUTHTABLE
REPLACE PTR BY PST UNTIL = ","
REPLACE PTR:PTR BY PST:PST UNTIL IN ALPHA6
REPLACE PTR BY PST FOR LNGTH WHILE > "0"
REPLACE PTR BY PST FOR LEFT:25 WHILE IN ACCEPTABLE
REPLACE PTR BY PST FOR 120 UNTIL NEQ " "
REPLACE PTR BY PST FOR M:N UNTIL IN ALPHA
REPLACE PTR:PTR BY SUMTOTAL FOR 6 DIGITS
REPLACE PTR BY FYLE.TITLE
REPLACE PTR BY PST:PST FOR L WITH XLATTABLE
```

Semantics

The general explanation of string handling found under the *<string statement>* should be read and understood before attempting to use the following information.

The *<replace statement>* causes character data from one or more data sources to be stored in a designated portion of an array row. The array row and the starting character position within the array row are both determined by the *<pointer expression>* part of the *<destination>*. The value of this *<pointer expression>* initializes the stack-destination-pointer. As each character is moved into the destination array row, the stack-destination-pointer is correspondingly incremented one character position. When the last character has been stored in the destination array row, the corresponding final value of the stack-destination-pointer is stored in the *<pointer variable>* of the *<update pointer>*, if the *<update pointer>* is not empty; otherwise, it is discarded.

The *<source list>* specifies the data and the processing to be performed upon this data to obtain the character data to be stored in the destination array row. The *<source list>* consists of one or more *<source part>s.* Each *<source part>* specifies source data and the processing to be performed upon the data. All the data specified by a single *<source part>* is processed by a single method, but the various *<source part>s* of the *<source list>* can specify a variety of processing methods.

B 7000/B 6000 Series hardware requires character size to be provided for destination-pointer expressions and for most classes of character transfer. Use of a **HEX ARRAY**, a **BCL ARRAY**, an **ASCII ARRAY**, or an **EBCDIC ARRAY** provides the character size for transfers into these types of arrays. When building an explicit **POINTER** from a non-character array, a character size should be provided. Failure to provide the correct character size, where required, results in a run-time error **(INVALIDOP).**

With certain forms of the *<source part>*, provisions are made to store the final value of the stack-source-pointer. With several *<source part>s* in a single *<replace statement>*, several "final values" for the stack-source-pointer arise. Corresponding to these final values are values of the stack-destination-pointer. These latter values are not accessible to the programmer. They serve as the initial values of the stack-destination-pointer for the processing of each next *<source part>*.

The *<source>* is the same syntactical construct encountered in the *<scan statement>*. The *<source>* contains a *<pointer expression>* that initializes the stack-source-pointer to a particular character position in an array row. The character size associated within this *<pointer expression>* must be the same as that character size associated with the *<pointer expression>* that initialized the stack-destination-pointer. If the *<update pointer>* of the *<source>* is not *<empty>*, the *<pointer variable>* specified by the *<update pointer>* is assigned the final value of the stack-source-pointer for this part of the data being processed.

Pragmatics

The formal syntax of the *<source part>* can be reduced to the following combinations:

<arithmetic expression>
<arithmetic expression> **FOR** *<arithmetic expression>*
<arithmetic expression> **FOR** *<arithmetic expression>* **WORDS**
<arithmetic expression> **FOR** *<arithmetic expression>* **DIGITS**
<arithmetic expression> **FOR * DIGITS**
<numeric convert part>

<source> **FOR** *<arithmetic expression>*
<source> **FOR** *<arithmetic expression>* **WORDS**
<source> **FOR** *<arithmetic expression>* **WITH** *<translate table>*

<source> **FOR** *<count part>* **WHILE** *<relational operator>* *<arithmetic expression>*
<source> **FOR** *<count part>* **UNTIL** *<relational operator>* *<arithmetic expression>*
<source> **FOR** *<count part>* **WHILE IN** *<truthset table>*
<source> **FOR** *<count part>* **UNTIL IN** *<truthset table>*

<source> **WHILE** *<relational operator>* *<arithmetic expression>*
<source> **UNTIL** *<relational operator>* *<arithmetic expression>*
<source> **WHILE IN** *<truthset table>*
<source> **UNTIL IN** *<truthset table>*

<source> **WITH** *<picture identifier>*

<pointer-valued attribute>

The remainder of the information pertaining to the *<replace statement>* is organized according to the above combinations.

The first four combinations of *<source part>* have the leading syntactical item of *<arithmetic expression>*. The exact structure of the *<arithmetic expression>* has an effect on how the item is actually created and handled to accomplish the intended operation. The three allowable structures are "short string" (a quoted string of characters equal to or less than 48 bits), a "long string" (a quoted string longer than 48 bits), and "non-string" (either *<variable>* or the result of arithmetic manipulations). For ease of reference, three pseudo **BNF** items of {short string}, {long string}, and {non-string} have been created and are used below in describing the first four combinations of *<source part>*.

Each {short string} is represented by a 48-bit operand, within which the specified bits/characters are left-justified and the remaining bits are filled to the right by appropriate repetitions of the {short string} Once the entire 48-bit operand is evaluated at compile-time, all traces of the character size of the {short string} are discarded. At execution-time, if the operand is used in a *<source part>* where individual characters are to be copied from the operand, the size of the characters copied is determined by the character size of the stack-destination-pointer. Thus, if the character size of the stack-destination-pointer and the character size of {short string} are not the same, the results are likely to be unexpected and undesired by the programmer.

Each {long string} is stored (at compile-time) in a portion of one of the special arrays (called **POOL ARRAYS**) created by the compiler for use at execution-time. A pointer is automatically created at execution-time that points at the beginning of the {long string}. The created pointer must have a character size appropriate for the specified string; therefore, a {long string} must have a character size of 4-, 6-, or 8-bits. **EBCDIC** (8-bit) is the default character size unless the compiler is instructed otherwise. (Refer to appendix D.)

<arithmetic expression>

{short string}. The stack-source-operand is initialized by the value of the {short string} (appropriately evaluated as described). The stack-integer-counter is initialized by the string length of the {short string}. Characters, of the size specified by the stack-destination-pointer, are copied from the 48-bit stack-source-operand to where the stack-destination-pointer indicates until the stack-integer-counter is decremented to zero. If all of the bits of the stack-source-operand are copied before completion of the copy process, the stack-source-operand is reused as required.

{long string}. The stack-source-pointer is initialized by the value of the pointer that points to the first character of the {long string} in a **POOL ARRAY**. The stack-integer-counter is initialized to the length of the {long string}; for example, 17 in the following: **"12345"4"FFBBCC"8"123456789"**. The specified number (as indicated by the initial value of the stack-integer-counter) of characters is copied from the {long string} to where the stack-destination-pointer indicates. That is, the {long string} is copied exactly once.

{non-string}. The *<arithmetic expression>* is appropriately evaluated into a 48-bit operand and used to initialize the stack-source-operand. The entire 48-bit value of the stack-source-operand is copied, exactly once, to where the stack-destination-pointer indicates. The character size of the stack-destination-pointer is irrelevant.

<arithmetic expression> **FOR** *<arithmetic expression>*

{short string} **FOR** *<arithmetic expression>*. The stack-source-operand is initialized by the value of the {short string}. The stack-integer-counter is initialized by the *<arithmetic expression>*. Characters of the size specified by the stack-destination-pointer are copied from the 48-bit stack-source-operand to where the stack-destination-pointer indicates until the stack-integer-counter has been decremented to zero. If the stack-source-operand is copied completely before the stack-integer-counter is decremented to zero, the stack-source-operand is reused as many times as required.

{long string} **FOR** *<arithmetic expression>*. The stack-source-pointer is initialized by the value of the created pointer that points to the first character in the {long string} in a **POOL ARRAY.** The stack-integer-counter is initialized to the value of the *<arithmetic expression>*. The specified number of characters are transferred to where the stack-destination-pointer indicates.

The integerized value of the *<arithmetic expression>* must not exceed the string length, or the resulting action is undefined. (Subsequent data in the **POOL ARRAY** could be transferred and, eventually, the end of the **POOL ARRAY** could be encountered, resulting in a **STRINGPROTECT** error condition.)

{non-string} **FOR** *<arithmetic expression>*. The *<arithmetic expression>* preceding the reserved word **FOR** is evaluated into a 48-bit operand and used to initialize the stack-source-operand. The stack-integer-counter is initialized by the value of the *<arithmetic expression>* following the reserved word **FOR.** Characters, of a size specified by the stack-destination-pointer, are copied from the stack-source-operand and stored where the stack-destination-pointer indicates. As each character is copied, the stack-integer-counter is decremented. Copying continues until the stack-integer-counter is decremented to zero. If more characters are to be copied than the 48-bit stack-source-operand can provide in a single use, the stack-source-operand is reused as required.

<arithmetic expression> **FOR** *<arithmetic expression>* **WORDS**

{short string} **FOR** *<arithmetic expression>* **WORDS.** The stack-source-operand is initialized by the value of the {short string}. The stack-integer-counter is initialized by the *<arithmetic expression>*. The stack-destination-pointer is advanced to the next nearest word boundary if it is not already pointing to a word boundary. The entire 48-bit stack-source-operand is copied to where the stack-destination-pointer indicates a number of times equal to the initial value of the stack-integer-counter.

{long string} **FOR** *<arithmetic expression>* **WORDS.** The stack-source-pointer is initialized by the value of the pointer that points to the first character of the {long string} in a **POOL ARRAY.** The stack-integer-counter is initialized to the value of the *<arithmetic expression>*. In this case, the data representation of the {long string} is placed, at compile time, in the **POOL ARRAY,** left-justified to a word boundary. The stack-destination-pointer is advanced to the next word boundary if it is not already at a word boundary. The represented data of the {long string} is copied, 48 bits at a time, to where the stack-destination-pointer indicates, until the specified number of words are transferred. If the number of words specified by the stack-integer-counter exceeds the data represented by the value of {long string} , the resulting action is undefined. (Possibly, subsequent data in the **POOL ARRAY** would be copied and, eventually, the end of the **POOL ARRAY** would be encountered, resulting in a **STRINGPROTECT** error condition.)

{non-string} **FOR** *<arithmetic expression>* **WORDS**. The *<arithmetic expression>* preceding the reserved word **FOR** is evaluated into a 48-bit operand and used to initialize the stack-source-operand. The stack-integer-counter is initialized by the value of the *<arithmetic expression>* following the reserved word **FOR**. The stack-destination-pointer is advanced to the next word boundary, if it is not already pointing to a word boundary. The value of the stack-source-operand is copied to where the stack-destination-pointer indicates a number of times specified by the initial value of the stack-integer-counter. The character size of the stack-destination-pointer is irrelevant.

<arithmetic expression> **FOR** *<arithmetic expression>* **DIGITS**

The absolute value of the first *<arithmetic expression>* is evaluated, integerized to a single-precision operand, and used to initialize the stack-source-operand. The second *<arithmetic expression>* is evaluated, integerized to a single-precision operand, and used to initialize the stack-integer-counter. The value of the stack-source-operand is first transformed into a sequence of 12 decimal 4-bit characters. The value of the stack-integer-counter specifies how many of these decimal 4-bit characters (taken from the right-hand side) are to be selected for further transformation. If the stack-destination-pointer has a character size of 4, the selected characters are copied without further transformation to the destination array row. If the stack-destination-pointer has a character size of 6 or 8, either an appropriate 2-bit zone field is added to each character or an appropriate 4-bit zone field is added to each character before being copied to the destination array row. The 2-bit zone field is 1"00", and the 4-bit zone field is 1"1111". If the stack-integer-counter has a value greater than 12, an **INVALIDOP** occurs. If the value of the stack-integer-counter is not large enough to include all non-zero decimal characters, the overflow flip-flop is set. This flip-flop can be tested by the Boolean intrinsic function whose name is **OVERFLOW**.

The sign of the first *<arithmetic expression>* is placed in the external sign flip-flop. The significance of the value of the external sign flip-flop is explained in the discussion of the *<picture declaration>*.

The remaining combinations of *<source part>* have *<source>* as the first syntactical item. The syntax of *<source>* shows that a pointer with an optional *<update pointer>* is used to select the characters to be picked up and appropriately manipulated. The selected characters are either 4-, 6-, or 8-bits each, depending on the character size of the source pointer. The manipulation depends on the particular syntax used as well as the character sizes of the source and destination pointers. With the exception of the *<translate part>*, a mismatch between the source and destination character sizes will produce an invalid-op interrupt.

<arithmetic expression> **FOR * DIGITS**

The absolute value of the *<arithmetic expression>* is evaluated and integerized to a single-precision operand. This operand is then transformed into a sequence of 12 decimal 4-bit characters. Leading zeros in this operand are ignored, and the remaining characters are transferred to the destination. If the destination pointer has a character size of 4 bits, the digits are transferred unmodified. If the character size is 6 or 8, the characters are augmented with zone bits of 1"00" or 1"1111", respectively. The overflow flip-flop is set if the *<arithmetic expression>* exceeds 12 digits in absolute value; the external sign flip-flop is set if the *<arithmetic expression>* is negative.

<numeric convert part>

The *<arithmetic expression>* is evaluated. An intrinsic procedure is called to generate an **EBCDIC** character string representing the decimal value of the *<arithmetic expression>*. If the destination pointer has a character size of 8 bits, the resulting string is copied to the destination. If the character size is 6 bits, the string is copied with **EBCDIC-to-BCL** translation. If the character size is 4 bits, a fatal run-time error occurs.

If a *<count part>* appears, it specified the maximum field width to be used; if the * appears, the field is unlimited (but never exceeds 36 characters). If a *<residual count>* appears, the <simple variable> is assigned the difference between the specified maximum field and the characters actually used.

The form of the decimal representation is determined by the operand type (single/double), whether or not the the operand value is integral, the magnitude of the operand, the number of significant digits in its decimal representation, and upon the field width. The basic rule is the number will be represented as compactly as possible, using integer, decimal-point or scientific notation as appropriate.

For example, the following *<numeric convert part>s* generate the decimal representations given:

123 FOR * NUMERIC	**123**	
12345678 FOR 8 NUMERIC	**12345678**	
12345678 FOR 6 NUMERIC	**1.23+7**	
123/100 FOR N:6 NUMERIC	**1.23**	**(N:=2)**
1@@0/3 FOR * NUMERIC	**0.333333333333333333333**	

<source> **FOR** *<arithmetic expression>*

The stack-source-pointer is initialized to the source pointer. The stack-integer-counter is initialized to the value of the *<arithmetic expression>*. The specified number of characters are transferred to where the stack-destination-pointer indicates.

<source> **FOR** *<arithmetic expression>* **WORDS**

The stack-source-pointer is initialized to the source pointer. The stack-integer-counter is initialized to the value of the *<arithmetic expression>*. Both the stack-source-pointer and stack-destination-pointer must point at a word boundary. Either or both are automatically adjusted forward to word boundaries if necessary. The specified number of 48-bit words are transferred to where the stack-destination-pointer indicates.

<source> **FOR** *<arithmetic expression>* **WITH** *<translate table>*

The function of this construct is to retrieve characters from a source location, translate each such character (through the use of the specified translation table) into a possibly different character having a possibly different character size, and store each resulting character where the physical-destination-pointer indicates.

The value of the *<pointer expression>* points to the first character to be translated. The stack-source-pointer is initialized to the *<pointer expression>*. In the instance of the translation process, it is not required that the stack-destination-pointer and the stack-source-pointer have the same character size. Instead, the stack-source-pointer must have a character size equal to that of the characters in the array row being translated, and the stack-destination-pointer must have a character size equal to that of the resulting translated characters.

The *<arithmetic expression>* indicates the number of characters to be translated. The stack-integer-counter is initialized by the *<arithmetic expression>*. The stack-auxiliary-pointer is initialized to a pointer that indicates the location of the first word of a table to be used in the translation process. This pointer is derived from the *<translate table>*; it always points to the first character of the first word of the translation table, and its character size is absent. Normally, when a pointer value is used and its character size is absent, a default value of 8 is used. However, the character size of the pointer used to initialize the stack-auxiliary-pointer is irrelevant. The translation table is not examined sequentially, a character-at-a-time, but rather the data in the table is accessed by special indexing techniques implemented by hardware logic.

INTRINSIC TRANSLATION TABLES. If the *<translate table>* is of the form **ASCIITOBCL, ASCIITOEBCDIC, ASCIITOHEX, BCLTOASCII, BCLTOEBCDIC, BCLTOHEX, EBCDICTOASCII, EBCDICTOBCL, EBCDICTOHEX, HEXTOASCII, HEXTOBCL,** or **HEXTOEBCDIC**, the stack-auxiliary-pointer is initialized to a pointer that points to the appropriately supplied translation table. The function of each translation table is deduced from the name, for example, **BCLTOEBCDIC** implies that the table is to be used in translating characters from **BCL** to **EBCDIC**.

<translatetable identifier>. If the *<translate table>* is of the form *<translatetable identifier>*, a translation table will have been created by the **ALGOL** programmer through the use of the *<translatetable declaration>*. A detailed discussion regarding the construction of a translation table through the use of the *<translatetable declaration>* is provided with the syntax description for the *<translatetable declaration>*.

<subscripted variable>. If the *<translate table>* is of the form *<subscripted variable>*, the **ALGOL** programmer is responsible for creating a properly structured translation table that is contained entirely in the array row and begins with the word in the array row indicated by the *<subscripted variable>*. (See Figure 4-1.)

The next four combinations of the *<source part>* cause movement of characters from the source to the destination until either the specified number of characters have been transferred or until a source character fails/passes the specified test. (Refer to the *<scan statement>*.)

NOTE

If the total number of specified characters are transferred, the **TRUE/FALSE** Flip-Flop is set to **TRUE**. It is set to **FALSE** if the transferring stopped due to the test (see the Boolean intrinsic **TOGGLE**), and the stack-source-pointer is left pointing at the character that failed/passed the test.

<count part>

The syntax of *<count part>* shows that an *<arithmetic expression>* is the starting value of the number of characters to be transferred. A programmer may choose to have the *<residual count>* non-*<empty>*, in which case the value of the remaining count would be stored into the specified *<simple arithmetic variable>* at the completion of the *<source part>*.

<source> **FOR** *<count part>* **WHILE** *<relational operator>* *<arithmetic expression>*

The stack-source-pointer is initialized to the source pointer. The stack-integer-counter is initialized to the starting value of the *<count part>*. Characters are then transferred from the source to the destination until either the stack-integer-counter is decremented to zero or a source character fails the test.

<source> **FOR** *<count part>* **UNTIL** *<relational operator>* *<arithmetic expression>*

The stack-source-pointer is initialized to the source pointer. The stack-integer-counter is initialized to the starting value of the *<count part>*. Characters are then transferred from the source to the destination until either the stack-integer-counter is decremented to zero or a source character passes the test.

<source> **FOR** *<count part>* **WHILE IN** *<truthset table>*

The stack-source-pointer is initialized to the source pointer. The stack-integer-counter is initialized to the starting value of the *<count part>*. Characters are then transferred from the source to the destination until either the stack-integer-counter is decremented to zero or a source character fails the test. (See the *<truthset declaration>* for further information regarding the *<truthset table>*.)

<source> **FOR** *<count part>* **UNTIL IN** *<truthset table>*

The stack-source-pointer is initialized to the source pointer. The stack-integer-counter is initialized to the starting value of the *<count part>*. Characters are then transferred from the source to the destination until either the stack-integer-counter is decremented to zero or a source character passes the test. (See the *<truthset declaration>* for further information regarding the *<truthset table>*.)

The next four combinations of the *<source part>* cause movement of source characters to the destination until a source character fails/passes the specified test. In the first two cases, the *<source character>s* are tested against bits [7:8] or [5:6] or [3:4] of the *<arithmetic expression>*, depending on the character size of the *<source>*. In all cases, the stack-source-pointer is left pointing at the character that failed/passed the test.

<source> **WHILE** *<relational operator>* *<arithmetic expression>*

The stack-source-pointer is initialized to the source pointer. Characters are then transferred from the source to the destination until a source character fails the test.

<source> **UNTIL** *<relational operator>* *<arithmetic expression>*

The stack-source-pointer is initialized to the source pointer. Characters are then transferred from the source to the destination until a source character passes the test.

<source> **WHILE IN** *<truthset table>*

The stack-source-pointer is initialized to the source pointer. Characters are then transferred from the source to the destination until a source character fails the test. (See the *<truthset declaration>* for further information regarding the *<truthset table>*.)

<source> **UNTIL IN** *<truthset table>*

The stack-source-pointer is initialized to the source pointer. Characters are then transferred from the source to the destination until a source character passes the test. (See the *<truthset declaration>* for further information regarding the *<truthset table>*.)

<source> **WITH** *<picture identifier>*

The character data specified by *<source>* (which must be a pointer) is processed under control of the picture specified by the *<picture identifier>*. Details regarding the formation of a picture and the associated effect the picture has upon the processing of character data are described under the *<picture declaration>* description.

<pointer-valued attribute>

The string of characters indicated by the *<pointer-valued attribute>* is copied to where the stack-destination-pointer indicates. The string of characters is formatted into the destination array row in a form suitable to serve in the *<replace pointer-valued attribute statement>* that changes the same kind of *<pointer-valued attribute>* (the character string ends with an 8“.”). For example, the following sequence of statements is valid:

```
REPLACE P BY F1.TITLE;
REPLACE F2.TITLE BY P;
```

where **P** is a *<pointer identifier>*, and **F1** and **F2** are *<file identifier>s*.

All *<pointer-valued attributes>* have a character size of 8. At run-time, if the physical-destination-pointer does not also have a character size of 8, an **INVALIDOP** error condition occurs.

If a *<pointer-valued attribute>* appears as a *<source part>* in a *<replace statement>*, that part of the statement is not implemented by in-line code. Instead, a call is made on an **MCP** procedure to complete this part of the *<replace statement>*.

REPLACE FAMILY-CHANGE STATEMENT

Syntax

<replace family-change statement> ::= **REPLACE** *<family designator>*
BY *<up or down>* *<simple source>*

<family designator> ::= *<file designator>* . **FAMILY**

<up or down> ::= *+ | *–

<simple source> ::= *<string>* |
<pointer expression>

Examples

```
REPLACE NETWORK.FAMILY BY *+ "PROCTOR7."
REPLACE DATACOLLECTORS.FAMILY BY *– PTRTOSTANAME
```

Semantics

Once a remote file is opened, an **ALGOL** program can add stations to the family of the remote file or delete stations from the family of the remote file. The *<replace family-change statement>* is the **ALGOL** language construct provided for this purpose. The *<family designator>* specifies (through the *<file designator>*) the file whose attribute is to be changed, and the attribute name **FAMILY** specifies which attribute is to be changed. If a station is to be added to the family, *<up or down>* is *+. If the station is to be deleted from the family, *<up or down>* is *–. The *<simple source>* specifies the **TITLE** of the station involved. The *<simple source>* (as a value for an attribute) having a string of characters as its value must terminate with a period (8"."). (See further details under *<replace pointer-valued attribute statement>*.)

Pragmatics

If in the *<replace family-change statement>* the *<simple source>* does not reference a valid station **TITLE** as specified in the current **NDL**-specified network, then after the completion of the *<replace family-change statement>* the following occurs:

a. *<file designator>*.**FAMILY** is unchanged.
b. *<file designator>*.**ATTERR** is **TRUE**.
c. An appropriate error message is displayed on the **SPO**.
d. The program continues.

The *<file designator>*.**ATTYPE** is also set appropriately. If *<upordown>* is *– and the *<simple source>* specifies a valid station as defined by the current **NDL** description, but the specified station is not currently a member of the **FAMILY**, then the *<replace family-change statement>* makes no change to the specified **FAMILY**, indicates no error condition (such a situation is not considered to be an error), and control passes to the next statement of the program.

If the remote file is closed and opened again, the family reverts to its **NDL**-specified value.

In-line code is not generated by the compiler for the *<replace family-change statement>*. Instead, a call is made on an **MCP** procedure to complete the desired function.

REPLACE POINTER-VALUED ATTRIBUTE STATEMENT

Syntax

<replace pointer-valued attribute statement> ::= **REPLACE** *<pointer-valued attribute>*
BY *<simple source>* |
REPLACE *<pointer-valued attribute>* **BY**
<pointer-valued attribute>

<pointer-valued attribute> ::= *<file designator>* *<disk row/copy specifications>*
. *<pointer-valued file attribute name>* |
<task designator> . *<pointer-valued task attribute name>*

<pointer-valued task attribute name> ::= **BACKUPPREFIX** |
FILECARDS |
NAME |
USERCODE |
CHARGECODE

Examples

REPLACE FYLE.TITLE BY "MASTER/PAYROLL."
REPLACE FILEID(COPY2).TITLE BY PTRTONAME
REPLACE TSK.NAME BY "SECOND/STACK."
REPLACE T.NAME BY TS.NAME

Semantics

The *<replace pointer-valued attribute statement>* is the construct provided to change the value of a pointer-valued attribute to a *<simple source>*. The *<simple source>* represents or references the set of characters that are to become the new value of the pointer-valued attribute.

Pragmatics

If the *<simple source>* is a *<string>*, then the last character of the *<string>* must be a period. The "effective" part of the *<string>* terminates with the first period in the string. A maximum string length is associated with each *<pointer-valued attribute>*. If the effective part of the *<string>* has a string length that is greater than the maximum value specified relative to the particular *<pointer-valued attribute>*, the new value of the *<pointer-valued attribute>* is the *<string>* truncated on the right to the required length.

If the *<simple source>* is a *<pointer expression>*, at execution time the *<pointer expression>* must point to (or into) an *<array row>*. The *<array row>* must contain the string of characters that are to become the new value of the *<pointer-valued attribute>*. The *<pointer expression>* must point to the first character of this string of characters. Starting with the first character, characters are included in the value of the pointer-valued attribute until a period is encountered, or until the maximum number of characters is included, or the end of the array row is encountered. The latter results in an error condition.

If a *<pointer-valued attribute>* is used in the source, the source attribute and the destination attribute must be the same.

In-line code is not generated by the compiler for the *<replace pointer-valued atttribute statement>*. Instead, a call is made on an **MCP** procedure to complete the desired function.

RESET STATEMENT

Syntax

<reset statement> ::= **RESET** (*<event designator>*)

Examples

RESET (EVNT)
RESET (EVNTARAY [INDX])

Semantics

The *<reset statement>* resets an event to the **NOT HAPPENED** state. It does not cause any other action.

Pragmatics

If a *<reset statement>* is used after a *<wait statement>* to restore the event, a "window" of time exists within which another task or tasks could cause the event. For this reason, a *<waitandreset statement>* or *<causeandreset statement>* might prove to be more useful.

RESIZE STATEMENT

Syntax

<resize statement> ::= **RESIZE** (*<array row>* ,
<arithmetic expression> *<retain old>*)

<retain old> ::= *<empty>* |
, **RETAIN**

Examples

RESIZE (ARAY, NEWSZ)
RESIZE (INPUTDATA, FYLE.MAXRECSIZE, RETAIN)
RESIZE (ARAY[2,*],5)

Semantics

RESIZE (*<array row>* , *<arithmetic expression>*) causes the size of the specified *<array row>* to be changed to the size specified by the *<arithmetic expression>*. Information in the "new" <array row> is undefined.

The **RESIZE** (*<array row>*, *<arithmetic expression>*, **RETAIN**) form resizes the array row, and the information in the "old" *<array row>* is transferred into the "new" *<array row>* until all the information is transferred or the end of the "new" *<array row>* is encountered.

An *<array row>* in a multi-dimensioned array can be resized.

REWIND STATEMENT

Syntax

<rewind statement> ::= **REWIND** (*<file designator>*)

Example

REWIND (FILEA)

Semantics

The *<rewind statement>* causes the referenced file to be closed. If the file is a paper tape or magnetic tape file, it is rewound. For disk files, the record pointer is reset to the first record of the file. The file buffer areas are returned to the system. The **I/O** unit remains under program control.

Restriction

On paper tape files, the *<rewind statement>* can be used only on input.

RUN STATEMENT

Syntax

<run statement> ::= **RUN** *<procedure identifier>* *<actual parameter part>* [*<task designator>*]

Examples

RUN SIMPL [TSK]
RUN DOOER (X,Y,Z, "ABCD") [TSKARAY[INDX]]

Semantics

The *<run statement>* initiates a procedure as an independent task. Initiation consists of setting up a separate stack, initializing parameters (by value only), and beginning the execution of its statements. The initiator resumes execution and both run in parallel. The procedure must be compiled separately and declared **EXTERNAL**. All *<actual parameter>s* must be call-by-value.

Unlike the *<process statement>*, which it resembles, there is no dependence upon the initiator. Thus there is no "critical block" and the initiator can even go to end-of-job while the external procedure continues.

The contents of the *<task designator>* are simply copied by the **MCP** such that the initiated procedure has its own task variable. Prior to initiation, the task attributes can be initialized as needed. (Refer to *<arithmetic task attribute>* and *<Boolean task attribute>*.)

Pragmatics

Note that arrays and files cannot be declared value; therefore, procedures with array or file parameters cannot be invoked with a *<run statement>*. Also, a procedure with *<pointer parameters>*, whether or not it is declared value, cannot be invoked with a *<run statement>*.

SCAN STATEMENT

Syntax

<*scan statement*> ::= SCAN <*source*> <*scan part*>

Examples

```
SCAN PTR WHILE = " "
SCAN PTR UNTIL NEQ 4"00"
SCAN PTR:PTR WHILE IN ALPHA
SCAN PTR UNTIL IN ALPHA6
SCAN PTR:PTR WHILE IN ACCEPTABLE[0]
SCAN PTR FOR 50 WHILE > "Z"
SCAN PTR:PTR FOR X:80 UNTIL = "."
SCAN PTR FOR RMNDR:960 WHILE NEQ 4"1D"
SCAN PTR:PTR FOR ZED:ZED WHILE IN ALPHA8
SCAN PTR FOR 80 UNTIL IN GOODSTUFF [5]
```

Semantics

The general explanation of string handling found under the <*string statement*> should be read and understood before attempting to use the following information.

The function of the <*scan statement*> is to examine a contiguous portion of character data in an array row, a character-at-a-time, in a left-to-right direction.

<*source*> is always a <*pointer expression*> and the updated pointer can be stored at the completion of the <*scan statement*>.

<*scan part*> is basically a testing operation to determine when to stop the <*scan statement*>. The programmer can specify that scanning is to stop after a given number of source characters or when a source character fails/passes a specified test.

NOTE

If the total number of source characters are completely scanned, the **TRUE/FALSE** Flip-Flop is set **TRUE**. It is set to **FALSE** if the scan terminated due to the test failing/passing (see the Boolean intrinsic **TOGGLE**), and the stack-source-pointer is left pointing at the character that failed/passed the test.

As can be seen in the syntax of <*scan part*> (see <*replace statement*>), a <*count part*> would be used when a maximum number of source characters are to be scanned. A programmer may choose to have the <*residual count*> non-<*empty*>, in which case the value of the remaining count would be stored into the specified <*simple arithmetic variable*> at the completion of the <*scan statement*>.

The syntax of <*condition*> shows that the <*relational operator*> specifies the test to use between the <*arithmetic expression*> and the source characters.

The most common form of the <*arithmetic expression*> is a one-character string; e.g., ".", 8"A", 6"/", 4"00". However, a {non-string} item is allowed which contains the character against which source characters are tested. In either case, the stack-source-operand is initialized with the comparant character in a right-justified format (bits [7:8], [5:6], or [3:4] depending on the character size of the <*source*>).

Pragmatics

The formal syntax of the <*scan statement*> can be reduced to the following combinations:

<*source*> **FOR** <*count part*> **WHILE** <*relational operator*> <*arithmetic expression*>
<*source*> **FOR** <*count part*> **UNTIL** <*relational operator*> <*arithmetic expression*>
<*source*> **FOR** <*count part*> **WHILE IN** <*truthset table*>
<*source*> **FOR** <*count part*> **UNTIL IN** <*truthset table*>

<*source*> **WHILE** <*relational operator*> <*arithmetic expression*>
<*source*> **UNTIL** <*relational operator*> <*arithmetic expression*>

<*source*> **WHILE IN** <*truthset table*>
<*source*> **UNTIL IN** <*truthset table*>

The remainder of the information pertaining to the <*scan statement*> is organized according to the above combinations. Since all combinations of the <*scan statement*> begin with <*source*>, each description of a combination begins with the assumption that the stack-source-pointer has been initialized to the source pointer.

The first four combinations of the <*scan statement*> cause source characters to be scanned (skipped over), one-at-a-time, until either the specified number of characters have been examined or a source character fails/passes the test.

<*source*> **FOR** <*count part*> **WHILE** <*relational operator*> <*arithmetic expression*>

The stack-integer-counter is initialized to the starting value of the <*count part*>. Characters are then scanned, one-at-a-time, until either the stack-integer-counter is decremented to zero or a source character fails the test.

<*source*> **FOR** <*count part*> **UNTIL** <*relational operator*> <*arithmetic expression*>

The stack-integer-counter is initialized to the starting value of the <*count part*>. Characters are then scanned, one-at-a-time, until either the stack-integer-counter is decremented to zero or a source character passes the test.

<*source*> **FOR** <*count part*> **WHILE IN** <*truthset table*>

The stack-integer-counter is initialized to the starting value of the <*count part*>. Characters are then scanned, one-at-a-time, until the stack-integer-counter is decremented to zero or a source character fails the test.

<source> **FOR** *<count part>* **UNTIL IN** *<truthset table>*

The stack-integer-counter is initialized to the starting value of the *<count part>*. Characters are then scanned, one-at-a-time, until the stack-integer-counter is decremented to zero or a source character passes the test.

The remaining four combinations of the *<scan statement>* cause source characters to be scanned (skipped over) until a source character fails/passes the test. If the source data does not contain a character which is being scanned for, the scan operation will eventually encounter the end of the array row and a **SEG ARRAY ERROR**. This error causes the program to be terminated unless the appropriate *<on statement>* has been provided.

<source> **WHILE** *<relational operator>* *<arithmetic expression>*

Characters are scanned until a source character fails the test.

<source> **UNTIL** *<relational operator>* *<arithmetic expression>*

Characters are scanned until a source character passes the test.

<source> **WHILE IN** *<truthset table>*

Characters are scanned until a source character fails the test. (See the *<truthset declaration>* for further information regarding the *<truthset table>*.)

<source> **UNTIL IN** *<truthset table>*

Characters are scanned until a source character passes the test. (Refer to the *<truthset declaration>* for further information regarding the *<truthset table>*.)

SEEK STATEMENT

Syntax

<seek statement> ::= SEEK (*<file designator>* [*<record number>*])
<record number> ::= *<arithmetic expression>*

Example

SEEK (FILEA [X+2*Y])

Semantics

The *<seek statement>* is used with randomly accessed disk files. It provides the means by which the buffer of a file can be filled in advance of an anticipated read or write on the record to which the *<record number>* points.

The *<file designator>* must not be a direct file or a member of a direct switch file.

SET STATEMENT

Syntax

<set statement> ::= **SET** (*<event designator>*)

Examples

SET (EVNT)
SET (EVNTARAY [INDX])

Semantics

The *<set statement>* sets an event to the **HAPPENED** state. It does not cause any other action; that is, the *<set statement>* does *not* activate a task or tasks waiting on the event.

SORT STATEMENT

Syntax

```
<sort statement> ::= SORT ( <output option> ,
                            <input option> ,
                            <number of tapes> ,
                            <compare procedure> ,
                            <record length> <size specifications> )
                            <restart specifications>
<output option> ::= <file designator> |
                    <output procedure>
<output procedure> ::= <procedure identifier>
<input option> ::= <file designator> |
                   <input procedure>
<input procedure> ::= <procedure identifier>
<number of tapes> ::= <arithmetic expression>
<compare procedure> ::= <procedure identifier>
<record length> ::= <arithmetic expression>
<size specifications> ::= <empty> |
                          <core size> |
                          <core size> <disk size> |
                          <core size> <pack size>
<core size> ::= , <arithmetic expression>
<disk size> ::= , <arithmetic expression>
<restart specifications> ::= <empty> |
                             [ RESTART = <arithmetic expression> ]
<pack size> ::= , PACK <size>
<size> ::= <empty> |
           <arithmetic expressions>
```

Examples

```
SORT (FILEOUT, FILEIN, 3, COMPEAR, 10)
SORT (OUTPROC, INPROC, NUMOFTAPES, COMPARER, RECSZ, CORESZ, DSKSZ)
    [RESTART = PARAM]
```

Semantics

The *<sort statement>* provides a means whereby data, as specified by the *<input option>*, can be sorted and returned to the program as specified by the *<output option>*. The order in which the data is returned is determined by the *<compare procedure>*.

<output option>

If a *<file designator>* is specified as the *<output option>*, the *<sort statement>* writes the sorted output on this file. Upon completion of the *<sort statement>*, the file is closed. If the file is a disk file with a non-zero **SAVEFACTOR**, it is closed and locked. The output file must not be open when it is passed to the sort by the program.

If an *<output procedure>* is specified as the <output option>, the <sort statement> calls on this procedure once for each sorted record and once to allow end-of-output action. This procedure must be untyped and must used two parameters. The first parameter must be call-by-value Boolean, and the second parameter must be a one-dimensional array with a constant (0) lower bound. The Boolean parameter is **FALSE** as long as the second parameter contains a sorted record. When all records are returned, the first parameter is **TRUE** and the second parameter must not be accessed.

An example of an *<output procedure>* is as follows:

```
PROCEDURE OUTPROC (B, A);
VALUE B;
BOOLEAN B;
ARRAY A [0];
IF B THEN CLOSE (FILEID, RELEASE) ELSE WRITE (FILEID, RECISIZE, A[*] );
```

<input option>

If a *<file designator>* is used as the *<input option>*, the records in that file are used as input to the *<sort statement>*. This file is closed after all of the file records are read by the *<sort statement>*. Disk files are closed with regular close action, and non-disk files are closed with release action. The input file must not be open when it is passed to the sort by the program.

If an *<input procedure>* is used as the *<input option>*, the procedure is called to furnish input records to the *<sort statement>*. This *<input procedure>* must be a Boolean procedure with a one-dimensional array, with a constant (0) lower bound as its only parameter. This procedure, on each call, either inserts the next record to be sorted into its array parameter or returns the value **TRUE**, which indicates the end of the input data.

When a **TRUE** is returned by the *<input procedure>*, the *<sort statement>* does not use the contents of the array parameters and does not call on the *<input procedure>* again.

An example of an *<input procedure>* that sorts **N** elements of array **Q** is as follows:

```
BOOLEAN PROCEDURE INPROC (A);
ARRAY A [0] ;
IF NOT (INPROC := (N := N-1) < 0)
    THEN A [0] := Q [N] ;
```

<number of tapes>

The *<number of tapes>* specifies the number of tape files that can be used, if necessary, in the sorting process. If the value of the *<arithmetic expression>* is 0, no tapes are used. If the value of the *<arithmetic expression>* is less than 3, three tapes are used. If the value of the *<arithmetic expression>* is 8 or more, a maximum of eight tapes is used. If the value of the *<arithmetic expression>* is between 3 and 8, the value specified is used.

<compare procedure>

The *<compare procedure>* is called by the *<sort statement>* to determine which of two records must be used next in the sorting process. The procedure must be a Boolean with exactly two parameters. Each of the parameters must consist of one-dimensional arrays with constant (0) lower bounds. The Boolean value that is returned by the procedure must be **TRUE** if the array given as the first parameter is to appear in the output before the array given as the second parameter.

As an example, the following procedure could be used for sorting in ascending sequence:

```
BOOLEAN PROCEDURE CMP (A, B);
ARRAY A, B [0];
CMP := A[0] < B[0];
```

The identifier **CMP** is **TRUE** if array A is less than array B. **CMP** is **FALSE** if array A is greater than or equal to array B. This results in the lower-valued array being passed to the output first. In the preceding example, word [0] is the control on which sorting is to be performed.

For the actual comparison, two strings might be compared according to the **EBCDIC** collating sequence, or by using a string relation, or an arithmetic comparison might be performed by using an arithmetic relation. Also, the user could compare on different "keys" or fields in the records. The comparison technique is determined entirely by the user.

<record length>

The *<record length>* represents the length, in words, of the largest item that is presented to the *<sort statement>*. If the value of the *<arithmetic expression>* is not a positive integer, the largest integer that is not greater than the absolute value of the expression is used; that is, a record length of 12 is used if an expression has a value of -12.995. If the value of the *<arithmetic expression>* is 0, the program terminates.

<size specfications>

The *<size specifications>* allow the programmer to specify the amount of main memory and the amount of disk storage that can be used.

The *<core size>*, if present, specifies the number of words of main memory that can be used. If the number is unspecified, a value of 12,000 words is assumed.

The *<disk size>*, if present, specifies the amount of disk storage in words that can be used. If the amount is unspecified, a value of 600,000 words of disk storage is assumed.

<restart specifications>

The *<restart specifications>* give the sort the ability to resume processing at the most recent checkpoint following the discontinuance of a program. It is necessary for the program to provide logic to restore and maintain stack variables, arrays, files, pointers, etc., that are defined for, and by, the program. In other words, the program must provide the means to restore everything that is necessary for the program to continue from the point of interruption. This may be either a simple or difficult task and is entirely program-dependent. The restart capability is implemented only for disk sort.

<pack size>

The *<pack size>* allows programmatic specification of temporary files created by the *<sort statment>* to be on system resource pack rather than head-per-track disk. If *<size>* is *<empty>*, 600000 words of pack storage is assumed.

Pragmatics

SORT MODE

The combination of the *<disk size>* entry and the *<number of tapes>* determines the sort mode as follows:

a. Number of tapes ≠ 0, disk size = 0; sort mode is tape only.
b. Number of tapes ≠ 0, disk size ≠ 0; sort mode is Integrated-Tape-Disk (ITD).
c. Number of tapes = 0, disk size ≠ 0; sort mode is disk only.
d. Number of tapes = 0, disk size = 0; sort mode is core sort.

RESTART PARAMETER VALUES

The sort inspects various bits of the *<arithmetic expression>* parameter to determine the course of action it is to take. To control the sort, individual bits and combinations of bits can be set by the program. The meaning of the various bits and the decimal values used in the *<arithmetic expression>* to represent various bit combinations are explained in the paragraphs that follow.

Bit Values

The value of the least-significant (rightmost) five bits of the *<arithmetic expression>* are passed to the sort to indicate desired action. The various bits and their meanings are as follows:

BIT	STATE	DESCRIPTION
0	ON	The program is restarting a previous sort. The sort tries to open its two disk files and obtain restart information. If it is successful in obtaining this information, the sort tries to continue from the last-known restart point.
0	OFF	The sort is starting from the beginning. If the sort is restartable and previous sort files with identical titles exist, they are removed and replaced by new sort files.
1	ON	The program is requesting a restartable sort. The sort saves its two internal files and can be restarted upon program request. If bit 2 is **ON**, bit 1 is set by default.
1	OFF	A normal sort is requested and no sort files are saved (unless bit 2 is **ON**, which sets bit 1 by default).
2	ON	The program is requesting a restartable sort and desires extensive error recovery (from **I/O** errors). With this option set, the sort attempts to back-track and remerge strings, as necessary, when **I/O** errors occur during the accessing of either of the two sort files. To use this option, the program must provide at least

BIT	STATE	DESCRIPTION
		three times as much disk space as required to contain the input data. If less disk space is provided, the sort emits an error message, changes to restartable-only mode, and continues the sort without further capability of back-tracking.
2	**OFF**	Recovery from internal errors is not requested.
3	---	Bit 3 has meaning only if a restartable sort is requested. The use of this option controls the sort during the stringing phase as the user input is being read by the sort. Use of this bit determines how the sort restarts (when a restart is requested) only if the restart occurs while the sort is in the stringing phrase.
3	**ON**	The program requires that the sort restart at the beginning of the user's input. It is the equivalent of starting an entirely new sort. In case the restarted sort passes from the stringing phase into the merge phase, it continues from the merge phase. This bit can be set during a restart, even if it is not initially set. Once set, it cannot be reset by subsequent restarts.
3	**OFF**	The program requires the ability to restart at the last restart point that occurred during the stringing phase. If the sort is still in the stringing phase, it skips over the records already processed and continues from the last restart point. If the sort is in the merge phase, it continues from the last merge phase restart point. Use of this option, that is, by not setting the bit, is normally less efficient because more strings are created during the stringing phase.
4	---	This bit is reserved for expansion and is not currently used by the sort.

ARRAYS IN SORT PROCEDURES

If one or more sort procedures (input, output, or compare) are used, all must have the same specification for their array parameters. That is, if one declares its array parameter as an **EBCDIC ARRAY**, then all must declare their array parameters as EBCDIC or the procedures will not be syntactically accepted.

In addition, when character arrays are used with a sort, the record length parameter is interpreted as the length in characters.

For more detailed information concerning use of the <*sort statement*>, refer to the B 6000 Series System Operation Guide Reference Manual, form 5001563.

SPACE STATEMENT

Syntax

<space statement> ::= **SPACE** (*<file designator>* , *<arithmetic expression>*)
<action labels or finished event>

Examples

SPACE (FYLE,50)
SPACE (FILEID, N) [LEONF:LPAR]
SPACE (FILEID,–3) [LEOF:LPAR]
SPACE (FILEID, A + B – C) [EVNT]

Semantics

The *<space statement>* is used to bypass input records without reading them. The value of the *<arithmetic expression>* determines the number of records to be spaced and the direction of the spacing. If the *<arithmetic expression>* is positive, the records are spaced in a forward direction; if negative, in the reverse direction.

When the *<space statement>* is used on output files, records are bypassed in a manner similar to input records. The *<space statement>* can be used as a *<Boolean primary>*.

The *<file designator>* must not be a direct file or a member of a direct switch file.

STRING STATEMENT

Syntax

<string statement> ::= *<replace statement>* |
<scan statement> |
<replace pointer-valued attribute statement> |
<replace family-change statement>

Examples

```
REPLACE PTR BY ...
SCAN PTR UNTIL ...
REPLACE FYLE.TITLE BY ...
REPLACE DATACOM.FAMILY BY *+ ...
```

Semantics

A *<string statement>* can be any one of the four *<statement>s* indicated in the syntax.

The *<replace statement>* can be used to move string data into an array row. Within a single *<replace statement>*, the string data to be moved into an array row can arise from several sources. Each of these sources can be any of several different types. A source can be another array row, a *<string>*, the value of an *<arithmetic expression>*, or the character data indicated by a *<pointer-valued attribute>*. Furthermore, as the character data is moved from a source to the destination, the characters can be translated or edited. Also, an *<arithmetic expression>* source can be treated as a binary integer and converted into the equivalent decimal number expressed as a string of numeric characters.

The *<scan statement>* can be used to examine character data located in an array row.

The *<replace family-change statement>* is the language construct provided to add data communication stations to a family of stations or to remove data communication stations from a family of stations.

The *<replace pointer-valued attribute statement>* is the language construct provided to assign character data to where the *<pointer-valued attribute>* indicates.

<string statement>s operate upon character data sequentially in a left-to-right fashion.

Pragmatics

Many of the B 7000/B 6000 Series Information Processing System instructions used in the implementation of the four *<string statement>s* require that certain data be placed in the stack prior to the execution of the individual instructions. During the execution of any one of these instructions, the associated stack data is modified. In describing how the various forms of the *<string statement>* function, it is convenient to discuss how the stack data is initialized, what changes are made in the stack data, and what is done with the stack data at the end of the *<statement>* execution. To that end, the subject stack data items must be given names so that they can be discussed easily. The names to be used in the following explanations are as follows:

a. Stack-source-pointer.
b. Stack-destination-pointer.
c. Stack-integer-counter.
d. Stack-test-character.
e. Stack-source-operand.
f. Stack-auxiliary-pointer.

The word "stack" has been chosen to allude to the fact that these parameters do not correspond to the logical elements in the extended **ALGOL** language, but rather that these parameters have a temporary existence in the stack while the statement is being executed. Not all of these parameters are required for each or any one *<string statement>*.

The stack-source-pointer, the stack-destination-pointer, and the stack-auxiliary-pointer have the same internal structure as the *<pointer variable>s* that the programmer can declare in a program. These stack parameters are initialized either from *<pointer expression>s* that exist in the structure of the *<string statement>* or from previous corresponding stack parameters.

The initial value of the stack-source-pointer points to the first source character to be used by the associated instruction. As the execution of the instruction progresses, the stack-source-pointer is modified to point to each subsequent source character. When the instruction is complete, the stack-source-pointer points to the first "unprocessed" character in the source data. (What the "process" is depends upon the particular form of the *<string statement>*.) This final value can be stored into a *<pointer variable>*, if the programmer chooses, or it can be discarded.

The initial value of the stack-destination-pointer points to the first destination character position to be used by the associated instruction. As the execution of the instruction progresses, the stack-destination-pointer is modified to point to each subsequent destination character position. When the instruction is complete, the stack-destination-pointer points to the first unfilled character position in the destination data. If in mid-statement, this final value, corresponding to the completed processing of one element in the source list, is used as the initial value of a subsequent instruction, corresponding to processing commencement of the next element in the source list in the same statement. If at the end-statement, this final value can be stored into a *<pointer variable>*, if the programmer chooses, or it can be discarded.

The initial value of the stack-auxiliary-pointer points to the first entry of a table of data to be used by the instruction in its execution. This table can be a translation table if the instruction is extracting characters from the source data, translating the characters to different characters (possibly containing a different number of bits per character), and storing the translated characters into the destination data string. This table can be a table of bits (one bit per character in the character set involved) that defines a character subset. (The characters associated with bits having a value of one (1) are in the subset, and the characters associated with bits whose values are zero (0) are not in the subset.) Several of the *<string statement>s* use such a table. Finally, this table can be a table containing instructions (of a special type), called a **"PICTURE"**, which describes how the source string data is to be edited before being stored in the destination string.

The stack-integer-counter, when required by particular forms of the *<string statement>*, is initialized by an *<arithmetic expression>* supplied in the *<string statement>* by the programmer. The value of this *<arithmetic expression>* is integerized by the instruction requiring this parameter. The stack-integer-counter has different meanings depending upon the particular form of the *<string statement>* involved. In some cases, the number of characters in a source string to be processed (number of characters translated, number of words moved, number of characters moved) is dictated solely by this parameter. (The number of numeric characters to be placed into the destination string, while converting the value of an *<arithmetic expression>* to character form, is also dictated by the stack-integer-counter.) However, in some forms of the *<string statement>* two controlling factors exist that dictate how many characters are to be processed from a source string. One factor is source-data-dependent, and is called a *<condition>*. The other factor is a maximum count supplied by the stack-integer-counter and is initialized by an *<arithmetic expression>* supplied in the *<string statement>*. With such a *<string statement>* one could say, for example: "translate characters from the source string to the destination string until either 14 characters have been transferred or a period is encountered in the source string, whichever comes first." What actually happens is the following: the stack-integer-counter is initialized with the value of *<arithmetic expression>*; as each character is processed, the stack-integer-counter is decremented; the process stops when either the *<condition>* is satisfied (a period encountered, for example) or the count equals zero; the final value of the stack-integer-counter is available for storage if the programmer chooses to store it; otherwise, the final value is discarded; the syntactical element specifying where this final value is to be stored is the *<residual count>*.

The stack-test-character is initialized by an *<arithmetic expression>*, (usually, but not necessarily, of the form of a single-character string, such as **"B"**.) Although the stack-test-character parameter is one entire B 7000/B 6000 word, which contains the single precision value of the *<arithmetic expression>*, only the right-most character position is used. When a *<condition>* employing a *<relational operator>* is used in a *<string statement>*, the stack-test-character must contain the character against which the individual characters in the source string are compared. Several constructs in the B 7000/B 6000 Extended **ALGOL** Language cause the value of the **TRUE-FALSE** flip-flop to be established, that is, either set or reset. Those forms of the *<string statement>* that involve both a *<condition>* and a maximum count are among those constructs. At the end of each portion of a *<string statement>*, which concerns a single body of source data, and contains both a *<condition>* and a maximum count, the **TRUE-FALSE** flip-flop is set to **TRUE** if all of the data specified by the maximum count has been processed. This flip-flop is set to **FALSE** if not all of the data specified by the maximum count has been processed, that is, the *<condition>* stopped the processing. (Recall that the value of the **TRUE-FALSE** flip-flop is returned as the functional value of the Boolean instrinsic function **TOGGLE**.) Obviously, if a *<string statement>* involves several bodies of source data which, when processed, established the value of the **TRUE-FALSE** flip-flop, only the last established value can be obtained by the subsequent use of the **TOGGLE** function.

The stack-source-operand is used when the source data is represented by the value of an *<arithmetic expression>* rather than located in an array row that is pointed into by the stack-source-pointer. The stack-source-operand occupies the same position in the stack that the stack-source-pointer would otherwise occupy, and is initialized by the *<arithmetic expression>*.

Refer to the information under the specific *<string statement>s* for more detailed information.

SWAP STATEMENT

Syntax

<*swap statement*> ::= **SWAP** (<*array identifier*> , <*array identifier*>)

Examples

SWAP (ARAYA, ARAYB)
SWAP (DIRECTARAY1, DIRECTARAY2)

Semantics

The <*swap statement*> causes two multi-dimensional arrays to be interchanged. Note that they both must be multi-dimensional.

Pragmatics

The arrays must have the same length, character size, and number of dimensions, and can be direct or non-direct.

Attempting to mix direct and non-direct arrays is not allowed.

The two arrays must both belong to the same task.

THRU STATEMENT

Syntax

<thru statement> ::= **THRU** *<arithmetic expression>* **DO** *<statement>*

Examples

```
THRU 255 DO ...
THRU 2*LIMIT DO ...
THRU MAXI := REAL(PTR,3) DO ...
```

Semantics

The iterative *<thru statement>* is executed as follows:

The absolute value of the *<arithmetic expression>* is evaluated and integerized. This value indicates the number of times the *<statement>* following **DO** is to be executed. The upper limit is 2**39-1. Figure 5-6 illustrates the **THRU** loop.

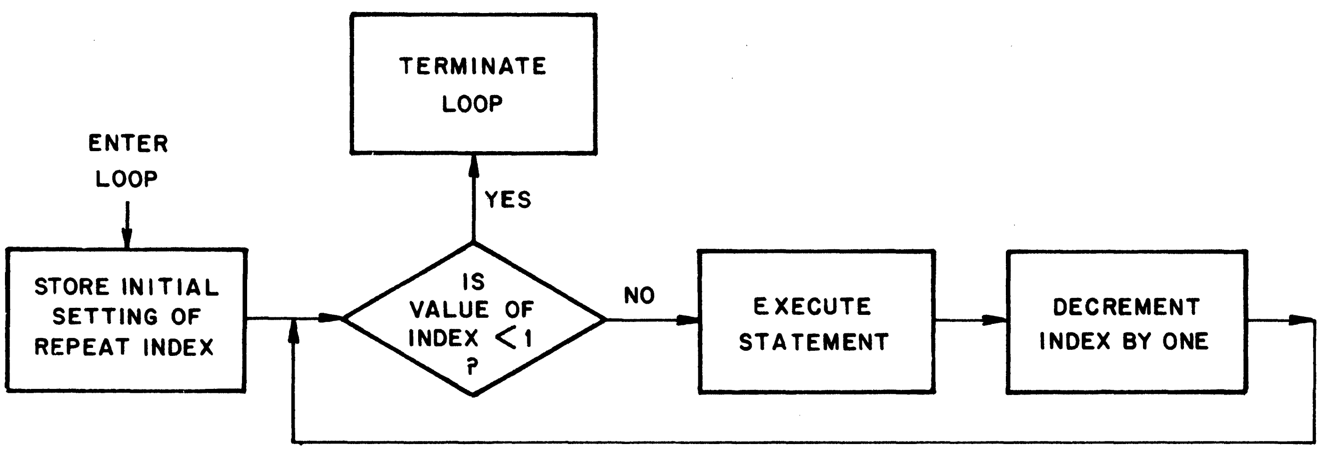


Figure 5-6. **THRU** Loop

UNCONDITIONAL STATEMENT

Syntax

<unconditional statement> ::= *<empty>* |
<assignment statement> |
<block> |
<breakpoint statement> |
<case statement> |
<changefile statement> |
<checkpoint statement> |
<compound statement> |
<continue statement> |
<deallocate statement> |
<event statement> |
<exchange statement> |
<fill statement> |
<go to statement> |
<**I/O** *statement*> |
<interrupt statement> |
<invocation statement> |
<iteration statement> |
<merge statement> |
<multiple attribute assignment statement> |
<on statement> |
<programdump statement> |
<removefile statement> |
<replace statement> |
<resize statement> |
<sort statement> |
<string statement> |
<swap statement> |
<vectormode statement> |
<when statement> |
<zip statement>

Semantics

The first choice of *<unconditional statement>s* is *<empty>*. This is referred to as a "dummy statement" since nothing is actually performed. For example, several *<label identifiers>* could be grouped as:

L1: L2: L3: . . .

which is legal syntax since the intervening *<statement>s* are *<empty>*. Furthermore, it is sometimes easier to do Boolean testing "backwards" such as:

IF A=B THEN ELSE X := X+1

Note that this example is to show *<empty>* as an *<unconditional statement>* and not good **ALGOL** programming.

The syntax for an *<unconditional statement>* is recursive; a *<statement>* can be a *<block>* or a *<compound statement>*, each of which in turn is composed of *<statement>s*.

VECTORMODE STATEMENT

Syntax

<vectormode statement> ::= **DO VECTORMODE** (<increment part> <vector part> **FOR**
<arithmetic expression>) <vectormode compound statement>

<increment part> ::= <empty> |
[<vector increment>] , |
[<vector increment>,<vector increment>] , |
[<vector increment>,<vector increment>,
<vector increment>] ,

<vector increment> ::= <arithmetic expression>

<vector part> ::= <vector reference>, |
<vector reference>, <vector reference>, |
<vector reference>, <vector reference>, <vector reference>,

<vector name> ::= <array row> |
<subscripted variable>

<vector reference> ::= <vector name> |
<vector identifier> = <vector name>

<vector identifier> ::= <identifier>

<vectormode compound statement> ::= **BEGIN** <vector compound tail> |
BEGIN <label declaration> <vector compound tail> |
<vector statement>

<vector compound tail> ::= <vector statement> **END** |
<vector statement> ; <vector compound tail>

<vector statement> ::= <conditional statement> |
<go to statement> |
<assignment statement> |
<exit statement> |
<increment statement>

<exit statement> ::= **EXIT**

<increment statement> ::= **INCREMENT** <address list>

<address list> ::= <vector address> |
<vector address> , <address list>

<vector address> ::= <vector name> |
<vector identifier>

Examples

```
DO VECTORMODE (ARRID1[*], ARRID2[2, *],
    ARRID3 [K-2, I], FOR 100) BEGIN EXIT END;
DO VECTORMODE (ARRID1[*],AA=AARID1[I], FOR X * A) BEGIN
    INCREMENT ARRID1, AA END;
DO VECTORMODE (ARRID1[*], FOR N) BEGIN EXIT END;
DO VECTORMODE (ARRID1[K-1], RA[*], FOR N-1) BEGIN
    INCREMENT ARRID1, RA END;
DO VECTORMODE ([-1, -1, 1], A[*], B[*], C[*], FOR N) BEGIN. . . END;
```

Semantics

From one of three vectors are specified in the **DO VECTORMODE** statement.

The *<increment part>* allows from one to three *<vector increment>s*, as is allowed in extended vectormode. The <vector increment>*s* must be constant expressions with values of one or minus one when integerized. Omitted increments are assumed to be one for single-precision vectors and two for double-precision vectors.

If a *<vector increment>* is specified for a double-precision vector, increment an even number of times to prevent indexing into the middle of a value.

The *<vector name>* determines the vector's starting point in an array. The length of the vectors becomes a function of the value following **FOR**.

The *<vector name>* cannot refer to a segmented array. In order to use an array that is longer than 1023 words in vector mode, it must be declared **LONG**.

If more than one of the vectors is in the same array, the second and third must have distinguishing identifiers since within the *<vector compound statement>* the vectors will be referred to only by *<array name>* without subscripts, or by the *<vector identifier>* which is used to avoid ambiguity.

For example,

```
DO VECTORMODE (A[*], B=A[20], C=[40] FOR N) . . .
```

allows references to vectors A, B, and C, all within the array A.

The *<go to statement>* is interpreted in the following manner. If the label is local to the vector mode block, only a branch forward is allowed. If the label is outside the vector mode block, vector mode is exited and code is executed to branch to that label. Because all labels inside the *<vector compound statement>* are local, no branching is permitted into the range of that statement.

The *<exit statement>* specifies to exit vector mode and continue execution with the first executable statement following the *<vector compound statement>*.

The *<increment statement>* increments the address of the vector element currently referenced by one (1) for single-precision arrays and two (2) for double-precision arrays.

Pragmatics

Arithmetic expressions in vector mode are strictly limited in form. They must meet the following requirements.

a. Procedure calls of any sort are prohibited. This means any call on an intrinsic function that is not in-line (expressed or implied) is prohibited. For example, **LN** may not be called. Since exponentiation generally calls an intrinsic, non-constant, non-integer exponentiation is prohibited.
b. Reference to any pointers or character arrays is prohibited throughout vector mode and its invocation.
c. Reference to any file, call-by-name parameter at any level, array or subscripted variable (other than the *<vector identifier>s* themselves) is prohibited. Simple variables that are at any level or all call-by-value parameters may be referenced.

Any arithmetic or logical variable in the stack can be referenced. However, no reference may be made that might cause an interrupt. This means, in particular, that call-by-name parameters, files, events, tasks, and pointers may not be referenced.

It is more efficient to increment a vector address after a reference to it, rather than before. There are no implied increments in a *<vectormode statement>*. Thus, if no such statements appear, the vector addresses are never incremented. For example:

```
INCREMENT A, B;
```

would increment the address for vectors A and B.

```
INCREMENT A, A;
```

would increment the address for vector A twice.

WAIT STATEMENT

Syntax

```
<wait statement> ::= WAIT ( <wait parameter list> ) |
                     WAIT ( <direct array row> ) |
                     WAIT
<wait parameter list> ::= <event list> |
                          ( <time> ) , <event list> |
                          ( <time> )
<event list> ::= <event designator> |
                 <event list> , <event designator>
<time> ::= {the amount of time in seconds (fractional seconds allowed) }
```

Examples

```
WAIT (EVNT)
WAIT (EVNT1, EVNT2, EVNT3)
X := WAIT ((NAPTIME), WAKEUP, GOAWAY)
WAIT (DIRECTARRAY)
RSLT := WAIT (DIRINPUT)
WAIT
```

Semantics

The *<wait statement>* allows for the suspension of a task until: either a time period elapses or an event is caused, a previously initiated Direct *<I/O statement>* is finished, or a software interrupt occurs.

The *<wait statement>* and the *<waitandreset statement>* (using a *<wait parameter list>*) are identical except for the state to which the caused event is set during the cause process. If all tasks are waiting on the event via the *<wait statement>*, the state of the event is set to **HAPPENED**. If any one task is waiting on the event via the *<waitandreset statement>*, the state of the event is set to **NOT HAPPENED.**

The simplest form of WAIT (*<wait parameter list>*) is WAIT (*<event designator>*). When executed, the event is examined for being **HAPPENED** or **NOT HAPPENED.** If the event is **HAPPENED**, the *<wait statement>* is essentially a "no-operation." If the event is **NOT HAPPENED**, the task is suspended until the event is CAUSEd.

For the full WAIT (*<wait parameter list>*) syntax, a program is allowed to be suspended until any one event in the *<event list>* is caused or until the time as specified by the *<time>* element (in seconds) has elapsed. (Refer to the Pragmatics of the *<when statement>*.)

The WAIT (*<wait parameter list>*) form can be used as an integer function that returns a value, starting at 1, which represents the position in the *<wait parameter list>* of the item that caused the task to be activated. For example, in the statement:

```
T := WAIT ((.001), E1, E2);
```

the value of **T** is **1** if elapsed time caused the task to be activated; however, in the following example:

T := WAIT (E1, E2, E3);

the value of **T** is **2** if a cause on event **E2** activated the task. The implementation of this mechanism contains interlocks to guarantee that one and only one parameter can activate a task.

The form **WAIT** (*<direct array row>*) is one of the ways a task can determine if a previously initiated Direct <**I/O** *statement*> has finished. This form can also be used as a Boolean function, in which case the result descriptor of the **I/O** operation will be returned when the **I/O** is completed. (Refer to the B 6000 Series Operation Guide Reference Manual, form 5001563, for both format and meaning of the returned value.)

If the *<wait statement>* consists solely of **WAIT**, an **MCP** procedure is entered which suspends the task until an attached and enabled interrupt is invoked as a result of the associated event being CAUSEd. (Refer to *<interrupt declatation>*.)

WAITANDRESET STATEMENT

Syntax

<*waitandreset statement*> ::= WAITANDRESET (<*wait parameter list*>)

Examples

```
WAITANDRESET (EVNT)
WAITANDRESET (EVNT1, EVNT2, EVNTARAY [INDX])
WAITANDRESET ((.5), FINI, GOAWAY)
REASON := WAITANDRESET((SLEEPMAX),WAKEUP,LOOKAROUND)
```

Semantics

The <*waitandreset statement*> allows for the suspension of a program until the event is caused. It is identical to the <*wait statement*> except that the event is reset to the **NOT HAPPENED** state before resuming execution of the program.

WHEN STATEMENT

Syntax

<*when statement*> ::= **WHEN** (<*time*>)

Examples

WHEN (10)
WHEN (2*Y+Z)

Semantics

The execution of a <*when statement*> causes the **MCP** to suspend the processing of a program for the number of seconds specified by the <*arithmetic expression*> in parentheses. The number of seconds can be specified as either an integer or a fraction of a second.

Pragmatics

Depending on the amount of multiprocessing being performed and priorities of other programs in execution, the actual time that a program is suspended can vary widely in respect to the time specified by <*time*>, but it will be at least the <*time*> specified.

WHILE STATEMENT

Syntax

<*while statement*> ::= **WHILE** <*Boolean expression*> **DO** <*statement*>

Examples

WHILE TRUE DO . . .
WHILE INDX LEQ MAXVAL DO . . .
WHILE J:= J+1 LSS LIMIT DO SU := SVALUES[J]

Semantics

The iterative <*while statement*> is executed as follows.

The <*Boolean expression*> is evaluated and, if the result is **TRUE**, the <*statement*> following **DO** is executed. This sequence of events continues until the value is **FALSE**, or the <*statement*> following **DO** transfers control outside the <*iteration statement*>. Figure 5–7 illustrates the **WHILE-DO** loop.

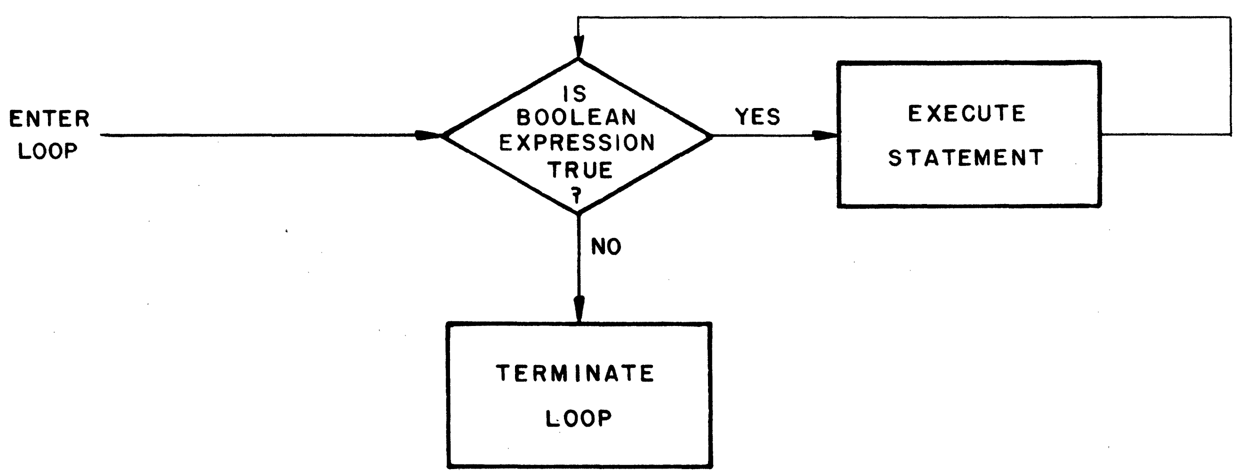


Figure 5–7. **WHILE-DO LOOP**

WRITE STATEMENT

Syntax

<write statement> ::= **WRITE** (*<write file part>* *<format and list part>*)
<action labels or finished event>

<write file part> ::= *<file part>* |
<task designator> . *<file-valued task attribute name>*

<file-valued task attribute name> ::= **TASKFILE**

Examples

```
WRITE (FILEID)
WRITE (SPOFILE, FMT, LISTID)
WRITE (FILEID [NO], FMT)
WRITE (SPOFILE, 10, ARRY[3,*])
WRITE (SWFILEID[0], X+Y-Z, ARRY[X,I,*])
WRITE (SPOFILE, /, LISTID)
WRITE (FILEID, FMT, LISTID)
WRITE (SWFILEID[3] [PAGE])
WRITE (FILEID, /, A,B,C)
WRITE (FILEID, SWFMT[A*I])
WRITE (FILEID, *, LISTID)
WRITE (FILEID [5+I], /, SWLISTID[4])
WRITE (FILEID, /, LISTID)
WRITE (FILEID, *, A,B,C)
WRITE (FILEID, FMT, A,B,C,D+SIN(X)) [:PARL]
WRITE (FILEID, FMT, LISTID) [:PARSWL[M]]
WRITE (SWFILEID[1], SWFMT[2], SWLISTID[3]) [:PARSWL[4]]
WRITE (DIRFYLE,30,DIRARAY) [EVNT]
WRITE (MYSELF.TASKFILE, <"ABOVE DUMP BEFORE TRANSACTION">)
```

Semantics

The *<write statement>* causes data to be transferred from various program variables to a peripheral device. The result of this *<statement>* depends on the form of the *<file part>* and the *<format and list part>*.

Because the syntax of the *<read statement>* and the *<write statement>* are identical, the user is referred to the semantics of the *<read statement>* for a discussion of the differences between the two statements.

ZIP STATEMENT

Syntax

<zip statement> ::= **ZIP WITH** *<array row>* |
ZIP WITH *<file designator>*

Examples

ZIP WITH ARAY
ZIP WITH FYLE

Semantics

The *<zip statement>* causes the **MCP** to activate the Work Flow Language compiler, using information in the *<array row>* or file referred to by the *<file designator>* as control cards and program parameter cards.

ZIP WITH *<array row>*

The information in the *<array row>* must appear as it normally would on punched cards; that is, as **BCL** or **EBCDIC** characters. The *<array row>* can be a **BCL** or **EBCDIC** string array row or a non-string array row. If the *<array row>* is not a string array, the character set expected by the **ZIP** intrinsic is determined by the setting of the **BCL $** option (refer to appendix D). The first character of the *<array row>* must be a question mark **(EBCDIC 4"6F" or BCL 3"14".)** The last "card" in the *<array row>* must contain the word **END** or **END JOB** followed by a period. The array row is processed as one punched card, but it can include more than 72 characters. A semicolon is used to separate control "cards" within the *<array row>*. Only one question mark character can appear in the *<array row>*.

The **MCP** examines the contents of the *<array row>* for correctness, and prints a message on the **SPO** if any errors are detected. If no errors are detected, the control information is obeyed. In either case, program control passes to the next statement in sequence.

ZIP WITH *<file designator>*

All control cards should be **EBCDIC** records that comply with standard B 7000/B 6000 Series Workflow Language (WFL) syntax. If records within the file are **BCL**, all records following the **<I> BCL** record should be **BCL**-coded up to and including the **<I> BCL** record or the first control card of the next deck for stacked decks. Following the first control card of the next deck, if any, all subsequent control cards must again be **EBCDIC**-coded.

Upon execution of a *<zip statement>*, the file referenced by the *<file designator>* is passed to the **MCP**. The program then continues processing in sequence.

NOTE

For both versions of the *<zip statement>*, refer to B 7000/B 6000 Series Workflow Language Reference Manual for further information on the format and content of the control cards.

6. EXPRESSIONS

EXPRESSION

Syntax

<expression> ::= *<arithmetic expression>* |
<Boolean expression> |
<case expression> |
<conditional expression> |
<designational expression> |
<function expression> |
<pointer expression>

Examples

```
X+Y
A=B
CASE N OF ( . . .
IF BOOL THEN . . . ELSE . . .
SWLBL[SWX]
SQRT( . . .
POINTER( . . .
```

Semantics

<expression>s are rules by which values can be obtained by executing various operations on the primaries of which *<expression>s* are composed. In the case of conditional and case *<expression>s*, the process is more complicated because one of several alternative *<expression>s* must first be selected for evaluation.

ARITHMETIC EXPRESSION

Syntax

<arithmetic expression> ::= <simple arithmetic expression> |
<final simple arithmetic expression> |
<conditional arithmetic expression>

<simple arithmetic expression> ::= <term> |
<unary operator> <term> |
<simple arithmetic expression> <adding operator> <term>

<term> ::= <factor> |
<term> <multiplying operator> <factor>

<factor> ::= <primary> |
<factor> ** <primary>

<primary> ::= <unsigned number> |
<string> |
<operand> <partial word part> |
<primary> & <arithmetic expression> <concatenation>

<operand> ::= <arithmetic variable> |
<arithmetic function designator> |
(<arithmetic expression>) |
<arithmetic case expression> |
<arithmetic attribute>

<arithmetic case expression> ::= <case head> (<arithmetic expression list>)

<arithmetic expression list> ::= <arithmetic expression> |
<arithmetic expression list> , <arithmetic expression>

<concatenation> ::= [<left bit-to> : <number of bits>] |
[<left bit-to> : <left bit-from> : <number of bits>]

<left bit-to> ::= <arithmetic expression>

<left bit-from> ::= <arithmetic expression>

<multiplying operator> ::= * | / | **DIV** | **MOD** | **MUX** | **TIMES**

<unary operator> ::= <adding operator>

<adding operator> ::= + | –

<final simple arithmetic expression> ::= <final term> |
<unary operator> <final term> |
<simple arithmetic expression>
<adding operator> <final term>

<final term> ::= <final factor> |
<term> <multiplying operator> <final factor>

<final factor> ::= <arithmetic variable> := <arithmetic expression> |
<arithmetic attribute> := <arithmetic expression> |
<factor> ** <arithmetic variable> := <arithmetic expression> |
<factor> ** <arithmetic attribute> := <arithmetic expression>

<conditional arithmetic expression> ::= *<if clause>* *<arithmetic expression>* **ELSE** *<arithmetic expression>*

Examples

FACTORS

VALID	INVALID
5.678	–9.81
CHARLIE	+DC8
(14+3.142)	B–A
	X*–3
	10–16

TERMS

VALID	INVALID
5.678	–13.6
MABEL	–(A+B)
KXF2	A+B
SUM/N	L*–A
(A+B)/(C–D)	*ENTIER (60)
2*(X+Y)	4(AC)

ARITHMETIC EXPRESSIONS

VALID	INVALID
COS (A+B)+C	3X + 4Y + Z
Y*3	A(X + 5)
+8	A + X*(B + X*(C + X*(D + X*E))))
(–B+SQRT(D))/(A+A)	P*[X + Y + Z]
–T*3	X + Y*–X + Z**2
5.678	
THETA	

Semantics

Arithmetic expressions yield numerical values by combining primaries in accordance with specified operations. The operators +, –, *, and / have the conventional mathematical meanings of addition, subtraction, multiplication, and division, respectively. Variables or function designators used as primaries in an *<arithmetic expression>* must be of an arithmetic *<type>*, that is, **REAL**, **INTEGER**, or **DOUBLE.**

PRECISION OF EXPRESSIONS

The value of an *<arithmetic expression>* can be expressed in single-or double-precision, depending upon the precision of its constituents or, in the case of **MUX**, the *<operator>* involved. The precision of an *<arithmetic expression>* is double if any *<variable>*, *<function expression>*, or *<number>* of which it is composed is of *<type>* **DOUBLE**, or if two terms are combined by the double-precision multiplication *<operator>* **MUX**. By examining the tag fields of the operands being combined, the hardware automatically extends the stack registers, when necessary, to accommodate extended precision numbers; thus special operators are not required. The **MUX** *<operator>* allows one to obtain a double-precision result from the multiplication of two single-precision operands.

The precision of a *<case expression>* value is double if any *<expression>* in its *<expression list>* is of *<type>* **DOUBLE**. Likewise, the precision of a *<conditional arithmetic expression>* is double if either *<arithmetic expression>* is double. In either case, single-precision arithmetic expressions are adjusted to double-precision, when necessary, by extension with zeros.

OPERATORS

The **DIV** operator denotes integer division. It has the following mathematical meaning:

Y DIV Z = SIGN(Y/Z)*ENTIER(ABS(Y/Z))

The **MOD** operator denotes remainder and has the following meaning:

Y MOD Z = Y–(Z*(SIGN(Y/Z)*ENTIER(ABS(Y/Z))))

The **MUX** operator multiplies either single- or double-precision operands, which yield a double-precision result. The ** operator denotes exponentiation. No two operators can be adjacent, and implied multiplication is not allowed. The **TIMES** operator denotes multiplication, as does the * operator.

PRECEDENCE OF *<arithmetic operator>s*

The sequence in which operations are performed is determined by the precedence of the operators involved. The order of precedence is as follows:

a. first: **
b. second: *, /, **MOD**, **DIV**, **MUX**, **TIMES**
c. third: +,–

When operators have the same order of precedence, the sequence of operation is determined by the order of the appearance, from left-to-right. Parentheses can be used in normal mathematical fashion to override the usual order of precedence.

Table 6–1. Operator Precedence

MATHEMATICAL EXPRESSION	EQUIVALENT ALGOL EXPRESSION	NON-EQUIVALENT ALGOL EXPRESSION
A X B	A * B	AB
$A + \frac{B}{2}$	A + B/2	
$\frac{X + 1}{Y}$	(X + 1)/Y	X + 1/Y
$\frac{D + E^2}{2A}$	(D + E **2)/(2 * A)	(D + E **2)/(2 A
$4(X + Y)^3$	4 * (X + Y) ** 3	4 * X + Y ** 3
$\frac{M - N}{(M + N)^{P}} + 5 \times 10^{-6}$	(M – N)/(M + N) ** P + 5 @ –6	

<primary>s

Parenthesized expressions are treated as *<primary>s;* that is, they are evaluated by themselves and the resulting value is subsequently combined with the other elements of the *<expression>*. Thus the normal precedence of operators can be overriden by the judicious placement of parentheses. Strings used as *<primary>s* must not exceed 48 bits in length; a *<string>* used as a *<primary>* is interpreted as either *<type>* **REAL** or *<type>* **INTEGER** depending upon its value.

EXAMPLES

```
5.678               +7
SIGMA               SIN X
Y1                  A/B
(X–Y)               –Z
COS(T)              +(X–Y)
ABS(a–X/Y)
((GEE+HAW)/PLOU)
(AX64*2+B)
```

EXPONENTIATION

The meaning of the double asterisks, exponentiation, **, depends upon the *<type>s* and the values of the primaries involved. Table 6–2 explains the various meanings of **Y**Z**.

Table 6–2. Exponentiation Meaning

	Z–TYPE INTEGER			Z–TYPE REAL		
	Z > 0	Z = 0	Z < 0	Z > 0	Z = 0	Z < 0
Y > 0	Note 1	1	Note 2	Note 3	1	Note 3
Y < 0	Note 1	1	Note 2	Note 4	1	Note 4
Y = 0	0	Note 4	Note 4	0	Note 4	Note 4

Note 1: Y**Z = Y*Y*Y . . . *Y(Z times)
Note 2: Y**Z = the reciprocal of Y*Y*Y . . . *Y(ABS(Z)times)
Note 3: Y**Z = EXP(Z*LN(Y))
Note 4: Value of expression is undefined.

<type>s OF RESULTING VALUES

The *<type>* of value resulting from an arithmetic operation depends on the *<operator>* and the *<type>* of operands being combined, except when the resulting value is undefined. Table 6–3 describes the *<type>* of quantity that results from various combinations of operands.

Table 6–3. Types of Values Resulting from an Arithmetic Operation

OPERAND ON LEFT	OPERAND ON RIGHT	+, –, *	/	DIV	MOD	**	MUX
INTEGER	INTEGER	Note 3	REAL	INTEGER	INTEGER	Note 1	DOUBLE
INTEGER	REAL	REAL	REAL	INTEGER	REAL	Note 2	DOUBLE
INTEGER	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	Note 2	DOUBLE
REAL	INTEGER	REAL	REAL	INTEGER	REAL	Note 2	DOUBLE
REAL	REAL	REAL	REAL	INTEGER	REAL	Note 2	DOUBLE
REAL	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE
DOUBLE	any	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE

Note 1: If the operand on the right is negative, or the absolute value of the result is greater than 2**39, REAL; otherwise, INTEGER.

Note 2: If the operand on the right is zero, INTEGER; otherwise, REAL.

Note 3: If the absolute value of the result is less than 2**39, INTEGER; otherwise, REAL.

The *<type>* of a *<case expression>* or a *<conditional arithmetic expression>* is **DOUBLE** if any of its constituent *<expression>s* are of *<type>* **DOUBLE (PRECISION OF EXPRESSIONS,** above). If the conditional or case *<expression>* contains any *<expression>s* of *<type>* **REAL,** its type is **REAL**; otherwise, its *<type>* is **INTEGER**, that is, a conditional or case *<expression>s* is of *<type>* **INTEGER** if and only if all its constituent *<expression>s* are of that *<type>*.

<concatenation>

The *<concatenation>* form of *<arithmetic expression>* provides an efficient method of forming a *<primary>* from selected parts of two or more *<primary>s.* A *<concatenation>* *<primary>* is formed by linking part of a *<primary>* with the specified portion of an *<arithmetic expression>* value. Since *<arithmetic expression>* is recursive with respect to *<concatenation>*, any number of *<concatenation>* terms can be used in constructing a *<primary>*.

The *<left bit-to>* part of *<concatenation>* term defines the leftmost bit location of the data field in the destination word. The *<left bit-from>* part defines the leftmost bit location of the data field in the source word. The *<number of bits>* part specifies the length of the data field to be moved from the source field to the destination field.

If the [*<left bit-to>* : *<number of bits>*] form of the *<concatenation>* term is specified, the source field is assumed to start at *<number of bits>* –1, that is, the source field is assumed to be the low order *<number of bits>* in the source word.

If more than one *<concatenation>* term is used in an *<expression>*, then these are evaluated from left-to-right.

Examples

The value of each of the succeeding expressions, when X=32767, Y=1024, and Z=1, is as follows:

EXPRESSION	VALUE
Y & (2*Z) [11:1:2]	2048
Y & Z [9:0:1] & X [3:13:4]	1551
Z & Y [40:10:2] & Z [45:0:1]	Floating point 1/64
X & Z [47:0:1]	32767
Y & (2*Z) [11:1:2] +5	2053
Y & (4*Z+1) [9:6:7] & X [14:14:15]	32767

<partial word part>

The *<partial word part>* term allows operations to be performed on any contiguous field within a word rather than the whole word. The *<left bit>* part defines the leftmost bit location of the field. The *<number of bits>* part specifies the length of the field.

The *<left bit-from>* part of a *<concatenation>* and the *<left bit>* part of a *<partial word part>* must lie within the range of 0 through 47, where bit 0 is the rightmost, or least-significant, bit in the word. The *<number of bits>* must lie within the range of 0 through 48. If *<number of bits>* exceeds the number of bits remaining in either the source or destination words, these fields are continued at bit number 47, leftmost, of the same word.

If through the use of *<variable>s* a program exceeds the starting bit number limits of 0 through 47 or the limit of *<number of bits>* of 0 through 48, an **INVALIDOP** will occur.

Examples

The value of each of the succeeding expressions, when X=32767, Y=2, and Z=4 (represented as integers), is as follows:

EXPRESSION	VALUE
X.[5:6]	63
IF Z.[3:2]=Y THEN 47 ELSE 23	23
((X+1)*Y*Z).[23:6]	1
X.[7:48]	4"FF000000007F"

BOOLEAN EXPRESSION

Syntax

<Boolean expression> ::= *<simple Boolean>* |
<final simple Boolean> |
<conditional Boolean expression>

<simple Boolean> ::= *<implication>* |
<simple Boolean> **EQV** *<implication>*

<implication> ::= *<Boolean term>* |
<implication> **IMP** *<Boolean term>*

<Boolean term> ::= *<Boolean factor>* |
<Boolean term> **OR** *<Boolean factor>* |
<Boolean term> | *<Boolean factor>*

<Boolean factor> ::= *<Boolean secondary>* |
<Boolean factor> **AND** *<Boolean secondary>*

<Boolean secondary> ::= *<Boolean primary>* |
NOT *<Boolean primary>*

<Boolean primary> ::= *<logical value>* |
<relation> |
<Boolean operand> *<partial word part>* |
<Boolean primary> & *<Boolean expression>* *<concatenation>* |
<table membership> |
<string relation> |
<pointer relation>

<logical value> ::= **TRUE** | **FALSE**

<relation> ::= *<arithmetic expression>* *<relational operator>* *<arithmetic expression>*

<Boolean operand> ::= *<Boolean variable>* |
<Boolean function designator> |
(*<Boolean expression>*) |
<Boolean case expression> |
<Boolean attribute>

<Boolean case expression> ::= *<case head>* (*<Boolean expression list>*)

<Boolean expression list> ::= *<Boolean expression>* |
<Boolean expression list> , *<Boolean expression>*

<table membership> ::= *<arithmetic expression>* **IN** *<table pointer>* |
<pointer expression> **IN** *<table pointer>*

<table pointer> ::= **ALPHA** | **ALPHA6** | **ALPHA7** | **ALPHA8** |
<truthset identifier> | *<subscripted variable>*

<string relation> ::= *<update pointer>* *<pointer expression>* *<relational operator>*
<update pointer> *<pointer expression>* **FOR** *<arithmetic expression>* |
<update pointer> *<pointer expression>* *<relational operator>* *<string>* |
<update pointer> *<pointer expression>* *<relational operator>* *<string>*
FOR *<arithmetic expression>*

<pointer relation> ::= *<pointer expression>* *<equality operator>* *<pointer expression>*

<equality operator> ::= = | **NEQ** | **EQL** | ¬=

<final simple Boolean> ::= *<final implication>* |
<simple Boolean> **EQV** *<final implication>*

<final implication> ::= *<final Boolean term>* |
<implication> **IMP** *<final Boolean term>*

<final Boolean term> ::= *<final Boolean factor>* |
<Boolean term> **OR** *<final Boolean factor>* |
<Boolean term> | *<final Boolean factor>*

<final Boolean factor> ::= *<final Boolean secondary>* |
<Boolean factor> **AND** *<final Boolean secondary>*

<final Boolean secondary> ::= *<Boolean assignment>* |
NOT *<Boolean assignment>*

<conditional Boolean expression> ::= *<if clause>* *<Boolean expression>*
ELSE *<Boolean expression>*

Examples

BOOLEAN PRIMARIES

VALID	INVALID
−1=0	A NEQ B OR C=D
1−A>B*(−E)	1−W*2
(X=Y OR W−K<4)	

BOOLEAN FACTORS

VALID	INVALID
X=0 AND Y≠0	A NEQ B OR C=D
A>1 AND (B=0 OR C<D)	1+A AND Z>0
(A=B OR C=D) AND (X<2 OR Y<2)	

BOOLEAN EXPRESSIONS

VALID	INVALID
A=B	(B*2−4XAXC)
I=0 AND J=0 OR K GEQ 1	

Semantics

Boolean expressions are rules for computing logical values. These expressions are analogous to arithmetic expressions in that they combine Boolean primaries according to fully recursive operations.

LOGICAL OPERATORS

The logical operators are defined by table 6–4.

Table 6–4. Truth Table

OPERAND A	OPERAND B	NOT A	A AND B	A OR B	A IMP B	A EQV B
TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE

The Boolean operations defined above are performed on all 48 bits of the copy(s) of the operand(s) involved, on a bit-by-bit basis. For example, **NOT TRUE** is not equivalent to **FALSE** because **NOT** complements all 48 bits of the constant **TRUE,** whereas all 48 bits of the constant **FALSE** are **OFF.**

PRECEDENCE OF *<logical operator>s*

First	:	*<arithmetic expression>s*
Second	:	*<relation>s*
Third	:	**NOT**
Fourth	:	**AND**
Fifth	:	**OR**
Sixth	:	**IMP**
Seventh	:	**EQV**

The *<logical operator>s* (**NOT**, **AND**, **OR**, **IMP**, **EQV**) are performed upon *<Boolean primary>s*. These *<Boolean primary>s* must be evaluated before they can be used as the operands upon which the *<logical operator>s* operate. For example, a *<relation>* is a *<Boolean primary>*. It consists of an *<arithmetic expression>*, a *<relational operator>*, and a second *<arithmetic expression>*. These two *<arithmetic expression>s* must be evaluated. Next, the truth of the stated relation between these two *<arithmetic expression>s* must be evaluated. This last evaluation produces a value of either **TRUE** or **FALSE,** and this value is the operand upon which subsequent *<logical operator>s* can act. When a group of *<Boolean primary>s* is connected by *<logical operator>s* forming a *<Boolean expression>*, the order in which the adjacent Boolean operands are combined by the intervening *<logical operator>* to form a new Boolean operand is determined by a precedence rule.

For this rule to be described easily, imagine parentheses around the *<Boolean expression>* of any *<Boolean assignment>*. Also imagine parentheses around the entire *<Boolean assignment>*. Note that these imagined parentheses make the subject *<Boolean expression>s* and *<Boolean assignment>s* (if any) into imagined *<Boolean operand>s* which are themselves *<Boolean primary>s*. The precedence rule then is: (1) evaluate a *<Boolean primary>* (real or imagined) before combining it with an adjacent *<Boolean primary>*, (2) apply **NOT** whenever it appears to the *<Boolean primary>* on its right forming a new operand, (3) these newly formed operands and the other *<Boolean primary>s* form the operands upon which the *<logical operator>s* **AND, OR, IMP,** and **EQV** operate, (4) if one of these operands is bounded on both sides by the same operator (from the group **AND, OR, IMP,** and **EQV**), then apply the operator on the left first, (5) if one of these operands is bounded on both sides by different operators (from the group **AND, OR, IMP,** and **EQV**), then apply the operator of higher precedence. The order is **AND, OR, IMP,** and **EQV**, with **AND** being the highest.

TABLE MEMBERSHIP

The *<table membership>* construct allows the programmer to test whether a given character is a member of a predefined table referenced by the *<table pointer>*. The character in question can be either a character in a *<string>* or a character in an *<array row>* referenced by a *<pointer expression>*.

The *<subscripted variable>* construct allows several tables to be contained in one *<array row>*, and the value of the *<subscript>* always indicates the beginning of the desired table. **ALPHA, ALPHA6,** and **ALPHA8** can be thought of as reserved *<subscripted variable>s,* **ALPHA8** is a *<table pointer>* for **EBCDIC** letters and digits; **ALPHA6** functions similarly for **BCL** letters and digits. **ALPHA** is the same as **ALPHA8** if the default *<character size>* has been specified as 8-bit; if the default *<character size>* is 6-bit, **ALPHA** is equivalent to **ALPHA6.**

(Refer to the *<truthset declaration>* for a description of how the *<table membership>* test references a bit in memory.)

<string relation>s

The *<string relation>* construct causes two pointers or a pointer and a *<string>* to be compared according to the **EBCDIC** collating sequence. The *<arithmetic expression>* specifies the number of characters to be compared, that is, the repeat count. If a literal *<string>* follows the *<relational operator>* and a repeat count has been specified, the *<string>* is concatenated with itself, if necessary, to form a 48-bit *<primary>*. The comparison is repeated until the repeat count is exhausted. If no repeat count is specified the string characters are compared once.

<pointer relation>s

A *<pointer relation>* determines whether two pointer expressions refer to the same character position in the same *<array row>*. If the *<character size>* of the two pointer expressions is unequal, the comparison always tests **FALSE.**

<concatenation>

The concatenation form of a *<Boolean expression>* provides an efficient means of forming a *<Boolean primary>* from selected parts of two or more *<Boolean primary>s.* It operates in the same way as the concatenation form of an *<arithmetic expression>*, except that it operates on *<Boolean primary>s.*

The construct *<relation>* & (*<relation>*) *<concatenation>* has not been implemented.

<partial word part>

The operation of the *<partial word part>* construct is directly analogous to that of the *<partial word part>* construct as described in the paragraphs on *<arithmetic expression>* and *<arithmetic assignment>*.

IS AND ISNT OPERATORS

The **IS** *<relational operator>* is used with arithmetic operands. It differs from the = sign in that it compares bit patterns for equality without doing normalization. For example, two **REAL** numbers, **R1** and **R2**, can have the same arithmetic value but different mantissas and exponents. In this case, the statement **R1 = R2** is a **TRUE** statement because normalization takes place before the comparison. However, the statement **R1 IS R2** is a **FALSE** statement because the comparison is done without normalization. The **ISNT** operator is the negation of the **IS** operator.

In addition, *<relational operator>s* other than **IS** and **ISNT** detect that a +0 hardware representation and a -0 hardware representation are equal. The same is true for a +0 and –0 exponent in the 48-bit hardware representation. Thus 4"400000000000" compared to 4"000000000000" tests **TRUE** if the = sign is used, but tests **FALSE** when using the *<relational operator>* **IS**.

CASE EXPRESSION

Syntax

<case expression> ::= *<case head>* (*<expression list>*)

<case head> ::= **CASE** *<arithmetic expression>* **OF**

<expression list> ::= *<arithmetic expression list>* |
<Boolean expression list>
<designational expression list> |
<pointer expression list>

Examples

```
CASE N OF (2, 20, 100, 37)
CASE X.[27:4] OF (TRUE,FALSE,TRUE,TRUE)
CASE TSTS[INDX] OF (LBL1,LBL2,AGAIN,NEXT,MORE)
CASE CHAR.SZF OF (PTR,PTS,POINTER(ARRAY),PTEMP,POLO)
```

Semantics

<case expression>s provide a convenient means of selecting one of many alternative expressions of the same kind to be evaluated at a particular point during the execution of a program. The *<expression>* to be evaluated is selected as follows: the *<arithmetic expression>* in the *<case head>* is evaluated and integerized by rounding if its value is not integral. This value is then used as an index into the *<expression list>*. The component expressions of the *<expression list>* are indexed sequentially from 0 through N-1, where N is the number of expressions in the list. The indexed *<expression>* is then evaluated and its value is the value of the *<case expression>*. If the value of the index lies outside the range 0 to N-1, an **INVALIDOP** interrupt occurs.

CONDITIONAL EXPRESSION

Syntax

<conditional expression> ::= *<conditional arithmetic expression>* |
<conditional Boolean expression> |
<conditional designational expression> |
<conditional pointer expression>

Examples

IF BOOL THEN 47 ELSE 95
IF A=B THEN BOOL ELSE FALSE
IF ALLDONE THEN EOJLBL ELSE NEXTLBL
IF CHAR.SZF=4 THEN PTRINEBCDIC ELSE PTRINBCL

Semantics

<expression>s of the form *<if clause>* *<expression>* **ELSE** *<expression>* are called *<conditional expression>s.* Depending upon either the value of the *<Boolean expression>* in the *<if clause>* or, if one or both of the alternative *<expression>s* are themselves conditional, the values of the *<Boolean expression>s* in several *<if clause>s*, or an *<expression>* are selected for evaluation. All alternative *<expression>s* must be of the same type.

The selection process proceeds as follows: first, the *<Boolean expression>* following the first *<if clause>* is evaluated; if the resulting value is **TRUE,** the *<expression>* following the **THEN** delimiter is evaluated, and the *<expression>* following the delimiter **ELSE** is ignored; otherwise, the *<expression>* following the **ELSE** delimiter is evaluated. If either of the alternative *<expression>s* is conditional, the process is repeated until an unconditional *<expression>* is selected for evaluation.

DESIGNATIONAL EXPRESSION

Syntax

<designational expression> ::= *<label designator>* |
<case head> (*<designational expression list>*) |
<conditional designational expression>

<label designator> ::= *<label identifier>* |
<switch label identifier> [*<subscript>*]

<conditional designational expression> ::= *<if clause>* *<designational expression>*
ELSE *<designational expression>*

<designational expression list> ::= *<designational expression>* |
<designational expression list> , *<designational expression>*

Examples

```
CHOOSEPATH[I + 2]
CASE X OF (GOTDATA,GOTERR,GOTREAL,GOTCHANGE,EXCADE)
IF K = 1 THEN SELECT[2] ELSE START
```

Semantics

A *<designational expression>* identifies a predefined label. As is true of other *<expression>s,* designational expressions may be differentiated as designational and *<conditional designational expression>s.*

<designational expression>

The process of evaluating a *<designational expression>* depends upon the constructs from which it is formed. If a *<designational expression>* is a *<label designator>*, the value of the expression is self-evident. When a *<designational expression>* is a *<switch label identifier>*, the actual numerical value of the subscript expression designates one of the elements in the *<switch list list>*. The element selected can be any form of the *<designational expression>* which is evaluated as stated above, or it can be a *<conditional designational expression>* which is evaluated as stated below.

If a *<designational expression>* is formed from a *<designational expression>* in parentheses, the latter is evaluated according to the applicable rules.

<conditional designational expression>

The evaluation of a *<conditional designational expression>* proceeds as follows:

a. The *<Boolean expression>* contained in the *<if clause>* is evaluated.
b. If a logical value of **TRUE** results, the *<designational expression>* following the *<if clause>* is evaluated, thus completing the evaluation of the *<conditional designational expression>*.
c. If the logical value produced by the *<if clause>* is **FALSE**, the *<designational expression>* following the delimiter **ELSE** is evaluated, thereby completing the evaluation of the *<designational expression>*.

Since the *<designational expression>s* following the delimiters **THEN** and **ELSE**, or both, can be *<conditional designational expression>s,* the analysis of the operation of a *<designational expression>* becomes recursive in a manner similar to that of the conditional arithmetic and *<Boolean expression>s.* However, in the case of a *<designational expression>*, the result produced is always a label.

<switch label identifier> [*<subscript>*]

The selection of the label in the *<switch label list>* is defined by positive integer values 1, 2, 3, . . . , N, where N is the number of entries in the *<switch label list>*. If the value of the *<subscript>* is of a *<type>* other than **INTEGER**, it is rounded to an integer in accordance with the rules applicable to the evaluation of *<subscript>s.* If the value of the *<subscript>* is outside the range of the *<switch label list>*, program control continues in sequence, without any error indication. Refer to *<switch label declaration>.*

BAD GO TO

Labels must be declared in, and therefore are local to, the innermost block in which they appear as a statement label. For example, a *<go to statement>* cannot lead from outside a block to a point inside that block; each block must be entered at the *<block head>* so that the associated declarations can be invoked.

A "bad go to" is the situation where a *<designational expression>* requires cutting back the lexicographic level; i.e., a branch of program control to a more global *<block>*. In order to accomplish a "bad go to", the **MCP** is invoked to discard any locally declared items which take memory space other than the stack itself (sometimes referred to as "non-stack items"), for example, locally declared files, arrays, and interrupts. Note that upon re-entry into that *<block>*, all those items would have to be re-established. For these reasons, it is best to declare frequently used non-stack items in the outermost *<block>*. Furthermore, frequently entered *<block>s* should only contain stack items. Failure to observe these rules will result in inefficient system use during program execution.

FUNCTION EXPRESSION

Syntax

<function expression> ::= *<arithmetic function designator>* |
<Boolean function designator> |
<pointer function designator>

Semantics

A function defines a single value which is the result of a specific set of operations on given parameters. Some functions are done "in-line" while others cause actual procedure entry. In either case, the value returned is a *<primary>* of the *<type>* specified for the function.

The *<actual parameter part>* must agree with the *<formal parameter part>* to which it corresponds both in the number and types of parameters. If the *<formal parameter>* is of *<type>* **INTEGER, REAL, ALPHA,** or **DOUBLE**, the *<actual parameter>* must be one of these four types, but not necessarily the same as its formal counterpart. If a mismatch occurs, the action that takes place depends upon whether the *<formal parameter>* is called-by-name or called-by-value. If it is called-by-value, the *<type>* of the *<actual parameter>* is converted to the *<type>* of the *<formal parameter>* before the latter is assigned its value. If the *<formal parameter>* is called-by-name, the appropriate conversion takes place each time the *<formal parameter>* is referenced.

ARITHMETIC FUNCTION DESIGNATOR

Syntax

<arithmetic function designator> ::= *<arithmetic function identifier>* *<actual parameter part>*

<arithmetic function identifier> ::= *<procedure identifier>* |
<arithmetic intrinsic name>

<arithmetic intrinsic name> ::= {a name of an arithmetic intrinsic "known" by the **ALGOL** compiler}

Arithmetic Intrinsic Names

The following arithmetic functions are intrinsic to **ALGOL**. For the purpose of *<type>* conversion, all arithmetic parameters are assumed to be called-by-value.

Parameters are *<type>* **REAL** or **INTEGER**, unless otherwise indicated. The following notation is used in the intrinsic list:

ABBREVIATION	MEANING
<ae>	*<arithmetic expression>*
<be>	*<Boolean expression>*
<pe>	*<pointer expression>*

(Refer to the **B 7000/B 6000** System Operation Guide Reference Manual, form 5001563, for a detailed description of the arithmetic intrinsics.)

FUNCTION	PARAMETER(S)	RESULT
ABS(*<ae>*)		Real. Absolute value of *<ae>*.
ARCCOS(*<ae>*)		Real. Principal value of arccosine of *<ae>*, −1 〈 *<ae>* 〈 1.
ARCSIN(*<ae>*)		Real. Principal value of the arcsine of *<ae>*, −1 〈 *<ae>* 〈 1.
ARCTAN(*<ae>*)		Real. Principal value of the arctangent of *<ae>*.
ARCTAN2(*<ae1>*,*<ae2>*)	real,real	Real. The principal value of the arctangent of *<ae1>* / *<ae2>*.
ATANH(*<ae>*)		Real. Hyperbolic arctangent of *<ae>*.

FUNCTION	PARAMETER(S)	RESULT
CHECKSUM(<*array row*>, <*ae1*>,<*ae2*>)		Real. Returns a hash function of all bits for words of the <*array row*> between <*ae1*> and <*ae2*> for use in parity checking.
		The hash function is generated by performing the logical equivalence function of each 48-bit array word and a 48-bit accumulator. The accumulator is cyclicly shifted left one bit at each step.
COMBINEPPBS(<*array row*>, <*array row*>)		Returns the new size of the second array. This function is used for combining program parameter blocks (PPBS). Each array is assumed to contain a PPB. The two arrays are combined with the second array taking precedence. The second array is resized, if necessary. As a result, the arrays cannot be direct arrays. The arrays must be Boolean, real, or integer non-read only arrays (i.e., no value arrays).
		Any attempt to access COMBINEPPBS except by a compiler results in the stack being ds-ed at execution time.
COMPILETIME(<*ae*>)	integer	Obtains various system time values, retained at compile-time, for use by the object program. The form of the value returned by the COMPILETIME intrinsic is the same as the TIME intrinsic for the same argument. The returned value is computed by the compiler, using the TIME intrinsic at compile-time. The argument must be a constant. (Refer to TIME intrinsic.)
	20	Allows the user to gain access to the compiler version number in integer form. COMPILETIME(20) DIV 10 yields the mark number. COMPILETIME(20) MOD 10 yields the level number.
	21	Gives the version cycle in integer form.
	22	Gives the patch number in integer form.
COS(<*ae*>)		Real. Cosine of <*ae*>.
COSH(<*ae*>)		Real. Hyperbolic cosine of <*ae*>.

FUNCTION	PARAMETER(S)	RESULT
COTAN(<ae>)		Real. Cotangent of <ae>.
DABS(<ae>)	double	Double. Absolute value of <ae>.
DAND(<ae1>,<ae2>)	double	Double. (<ae1>) and (<ae2>).
DARCCOS(<ae>)	double	Double. Principal value of the arccosine of <ae>, −1 〈 <ae> 〈 1.
DARCSIN(<ae>)	double	Double. Principal value of the arcsine of <ae>, −1 〈 <ae> 〈 1.
DARCTAN(<ae>)	double	Double. Principal value of the arctangent of <ae>.
DARCTAN2(<ae1>,<ae2>)	double, double	Double. The principal value of the arctangent of <ae1> / <a2>.
DCOS(<ae>)	double	Double. Cosine of <ae>.
DCOSH(<ae>)	double	Double. Hyperbolic cosine of <ae>.
DELTA(<pe1>,<pe2>)		Integer. The number of characters given by <pe2> minus <pe1>. Because DELTA must contend with such cases as segmented arrays, it is expensive in CPU time. (Refer to REAL (<pe>).)
DERF(<ae>)	double	Double. The value of the standard error function at <ae> ERF (−<ae>) = ERF (<ae>) at <ae>, poles at non-positive integers.
DERFC(<ae>)	double	Double. The complement of the value of the standard error function.
DEQV(<ae1>,<ae2>)	double, double	Double. (<ae1>) EQV (<ae2>).
DEXP(<ae>)	double	Double. e raised to the <ae> power.
DGAMMA(<ae>)	double	Double. The value of the gamma function at <ae>.
DIMP(<ae1>,<ae2>)	double, double	Double. (<ae1>) IMP (<ae2>).
DINTEGER(<ae>)		Returns a double-precision integer value equal to ENTIER(<ae> +0.5).

FUNCTION	PARAMETER(S)	RESULT
DLGAMMA(*<ae>*)	double	Double. The value of the natural logarithm of the gamma function at *<ae>*.
DLN(*<ae>*)	double	Double. Natural logarithm of *<ae>*.
DLOG(*<ae>*)	double	Double. Logarithm to base 10 of *<ae>*.
DMAX(*<ae1>*, . . .,*<aen>*)	double, . . . ,double	Double. Maximum of the values *<ae1>*, . . . ,*<aen>*: N 〉 1.
DMIN(*<ae1>*, . . . ,*<aen>*)	double	Double. Minimum of the values *<ae1>*, . . . ,*<aen>*: N 〉 1.
DNABS(*<ae>*)	double	Double. Negative DOUBLE absolute value of *<ae>*.
DNOT(*<ae>*)	double	Double. Complement of *<ae>*.
DOR(*<ae1>*,*<ae2>*)	double, double	Double. (*<ae1>*) OR (*<ae2>*).
DOUBLE(*<ae>*)		Returns the double value equal to the single REAL *<ae>*.
DOUBLE(*<ae1>*,*<ae2>*)		Returns the double value with first part equal to *<ae1>* and second part equal to *<ae2>*.
DOUBLE(*<update pointer>* *<pe>*, *<ae>*)		This function returns as an extended precision value, the decimal value represented by the string of characters starting with the character indicated by the pointer expression. The length of the string as determined from the arithmetic expression must be less than 24. A zone bit configuration of 1101 for 8-bit characters or 10 for 6-bit characters in the least-significant character position causes the result to be negative. With 4-bit characters, a 1101 in the most-significant character position yields a negative value. The state of the *<pointer expression>* at the exhaustion of the count can be preserved by an *<update pointer>*.
DSCALELEFT(*<ae1>*, *<ae2>*)	double, real	Double. Value of *<ae1>* multiplied by (10 raised to the *<ae2>* power).
DSCALERIGHT(*<ae1>*, *<ae2>*)	double, real	Double. Rounded result of *<ae1>* divided by (10 raised to the *<ae2>* power).

FUNCTION	PARAMETER(S)	RESULT
DSCALERIGHTT(<*ae1*>, <*ae2*>)	double, real	Double. Truncated result of <*ae1*> divided by (10 raised to the <*ae2*> power).
DSIN(<*ae*>)	double	Double. Sine of <*ae*>.
DSINH(<*ae*>)	double	Double. Hyperbolic sine of <*ae*>.
DSQRT(<*ae*>)	double	Double. The square root of <*ae*>.
DTAN(<*ae*>)	double	Double. Tangent of <*ae*>.
DTANH(<*ae*>)	double	Double. Hyperbolic tangent of <*ae*>.
ENTIER(<*ae*>)		Returns the largest integer not greater than the value of the arithmetic expression. The ENTIER function is frequently misunderstood to perform simple truncation. The following examples show that this is not the case. ENTIER (2.6) = 2; ENTIER (3.1) = 3; ENTIER (–0.01) = –1; ENTIER (–3.4) = –4; ENTIER (–1.8) = –2.
ERF(<*ae*>)		Real. The value of the standard error function at <*ae*> ERF(<–ae>) = –ERF(<*ae*>) at <*ae*>, pole at non-positive integers.
ERFC(<*ae*>)		Real. The complement of the value of the standard error function.
EXP(<*ae*>)		Real. ε (epsilon) raised to the <*ae*> power.
FIRSTONE(<*ae*>)		Integer. Bit number of leftmost nonzero bit in <*ae*>, plus one. It is set to 0 if no nonzero bit is found.
FIRSTWORD(<*ae*>)		Returns the first word of the double expression <*ae*> unchanged.
FIRSTWORD(<*ae*>, <*arithmetic variable*>)	real	Returns the first word of the double expression <*ae*> unchanged. It stores the second word of the expression <*ae*> in <*arithmetic variable*>.

FUNCTION	PARAMETER(S)	RESULT
GAMMA(*<ae>*)		Real. The value of the gamma function at *<ae>*.
INTEGER(*<ae>*)		Returns ENTIER(*<ae>*+0.5).
INTEGER (*<update pointer>* *<pointer expression>*, *<ae>*)		This function returns, as a single-precision integer, the decimal value represented by the string characters starting with the character indicated by the pointer expression. The largest integer value that can be returned is 549, 755, 813, 887. The length of the string as determined from the arithmetic expression must be less than 24. A zone bit configuration of 1101 for 8-bit characters or 10 for 6-bit characters in the least significant character position causes the result to be negative. With 4-bit characters, a leading 1101 results in a negative value.
INTEGERT(*<ae>*)		Integerizes the value of the arithmetic expression by truncation. For example, ENTIER(−1.2) = −2, while INTEGERT (−1.2) = −1.
LINENUMBER		Integer. Returns the sequence number of the card being read.
LISTLOOKUP (*<ae1>*, *<arrayrow>*, *<ae2>*)		A linked list of words is searched as follows: The *<array row>* is indexed by *<ae2>* and the word is extracted. Each word contains a value (in[47:28]) and a link (in [19:20]) into the next word. If the value in the extracted word is greater than or equal to *<ae1>*, the operation stops and the index to the word whose link points to the word with that value is returned. If the test fails, the link of the extracted word is used as an index in the *<array row>*; a new word is extracted and the process is repeated. A word with a link of zero terminates the list. Note that the value of a word is tested only if the link field is non-zero. If the linked list is exhausted (a word with a link of zero is encountered) a value of −1 is returned.
LN(*<ae>*)		Real. Natural logarithm of *<ae>*.

FUNCTION	PARAMETER(S)	RESULT
LNGAMMA(<ae>)		Real. The value of the natural logarithm of the gamma function at <ae>.
LOG(<ae>)		Real. Logarithm to base 10 <ae>.
MASKSEARCH (<ae1>,<ae2>,<. . .>)	<ae1>	<ae1> is the bit pattern to search for.
	<ae2>	<ae2> is the mask to be used in the search.
	<. . .>	<. . .> is an array that is a single-precision, single-dimensioned, nonsegmented array or an array row or subscripted variable. The following operation begins at the last word of the array (if no subscript is given) or at the specified word (if the subscript is given): A word is extracted and logically "ANDed" with <ae2> and compared (IS) with <ae1>. If the result IS <ae1>, the index of the extracted word is returned. If not, the index is decremented, and the operation is repeated until a match is found or the bottom of the array is encountered, in which case the result is –1.
MAX(<ae1>, . . . ,<aen>)	real, . . . ,real	Real. Maximum of the values <ae1>, . . . ,<aen>: N > 1.
MIN(<ae1>, . . . ,<aen>)	real, . . . ,real	Real. The minimum of <ae1>, . . . ,<aen>: N > 1.
NABS(<ae>)		Real. Negative absolute value of <ae>.
NORMALIZE(<ae>)	double	Double. Converts a double-precision operand to "rounded" single-precision operand.
ONES(<ae>)		Integer. Number of nonzero bits in <ae>.
POTL[<ae>] POTC[<ae>] POTH[<ae>]		Provides the value of 10**I, where I is an integer ($0 \leq I < 29604$), from three tables which are double precision read-only arrays. The value of 10**I can be computed using the arithmetic expression: POTL[I.[5:6]]*POTC[I[11:6]]* POTH[I.[14:3]]

FUNCTION	PARAMETER(S)	RESULT
POTL[*<ae>*] POTC[*<ae>*] POTH[*<ae>*] (Cont)		For example, RESULT := POTL[X] ; % WHERE X < 69 DEFINE POT (T) = (POTL[T.[5:6]] *POTC[T.[11:6]] * POTL[T.[14:3]])#; ONEDIVTENTOI := 1/POT(I);
RANDOM(*<ae>*)	real, call-by-name	Generates a random number between 0 and 1. The value of the argument is changed each time RANDOM is referenced.
READLOCK(*<ae>*, *<arithmetic variable>*)		Real. Taking only one memory cycle, *<ae>* is stored into the *<arithmetic variable>* and the prior content of the *<arithmetic variable>* is returned.
REAL(*<be>*)		Returns the real value represented by the *<Boolean expression>* *<be>*. All bits of *<be>* are used.
REAL(*<ae>*)	double	Returns the double expression *<ae>* normalized and rounded to a single.
REAL(*<pe>*)		Returns, as a real value, the value of the string descriptor *<pe>* with its tag SET to zero. (Refer to appendix B.)
REAL(*<pe>*,*<ae>*)		Returns, as a real value, a bit image of the string of *<ae>* characters starting with the character indicated by the pointer expression *<pe>*. All bits of each character are used.
SCALELEFT(*<ae1>*, *<ae2>*)		Integer. Value of *<ae1>* multiplied by (10 raised to the *<ae2>* power).
SCALERIGHT(*<ae1>*, *<ae2>*)		Integer. Rounded result of *<ae1>* divided by (10 raised to the *<ae2>* power).
SCALERIGHTF(*<ae1>*, *<ae2>*)		Left-justified packed decimal number (4-bit decimal) remainder of *<ae1>* divided by (10 raised to the *<ae2>* power). The number of significant digits returned is equal to *<ae2>*. The external sign FLIP-FLOP is set to the sign of *<ae1>* for use with the PICTURE editing phrases.

FUNCTION	PARAMETER(S)	RESULT
SCALERIGHTT(<*ae1*>, <*ae2*>)		Integer. Truncated result of <*ae1*> divided by (10 raised to the <*ae2*> power).
SECONDWORD(<*ae*>)	double	Returns the second word of the double expression <*ae*> unchanged.
SIGN(<*ae*>)		Integer. +1 if <*ae*> 〉 0, 0 if <*ae*> = 0, –1 if <*ae*> 〈 0.
SIN(<*ae*>)		Real. Sine of <*ae*>.
SINGLE(<*ae*>)	double	Returns the double expression <*ae*> normalized and truncated to a single.
SINH(<*ae*>)		Real. Hyperbolic sine of <*ae*>.
SIZE(< . . . >)	<*array designator*>	Integer. If a single-dimensioned array or a row of a multi-dimensioned array, it returns the size in units appropriate to the array, e.g. bytes, digits, or words.
		If a multi-dimensioned array, it returns the size of the first unspecified dimension.
	<*pointer identifier*>	The character size (4, 6, or 8) unless the <*pointer identifier*> is uninitialized, resulting in the value zero.
SQRT(<*ae*>)		Real. The square root of <*ae*>, <*ae*> 〉 0.
TAN(<*ae*>)		Real. Tangent of <*ae*>.
TANH(<*ae*>)		Real. Hyperbolic tangent of <*ae*>.
TIME(<*ae*>)	<*variable*>	TIME (<*ae*>) makes various system time values available.
		In-line code is generated for the cases where <*ae*> is an unsigned integer with one of the following values: 1, 4, 11, or 14. For all other cases, the value of <*ae*> is passed to an intrinsic function that yields the correct time as the result. Thus the code II:=1; J:= TIME(II); takes longer to execute than II:=1; J:= TIME(1);.

FUNCTION	PARAMETER(S)	RESULT
TIME(<ae>) (Cont)	0	Returns the current date in BCL characters (in the format: 6"YYDDD", where YY is the year and DDD is the day of the year).
	1	Returns as an integer value the time of day, in sixtieths of a second.
	2	Returns as an integer value the elapsed processor time of the program, in sixtieths of a second.
	3	Returns as an integer value the elapsed I/O time of the program, in sixtieths of a second.
	4	Returns as an integer value the contents of a 6-bit machine clock that increments every sixtieth of a second.
	5	Returns month, day, year as six BCL characters right-justified (in the format: 6"00MMDDYY").
	6	Returns a unique number for the time and date.
	7	Returns the year, month, day, hour, minute, second, and day of the week.
	10	Same as TIME(0), except time is expressed in EBCDIC characters (in the format 8"YYDDD").
	11	Same as TIME(1), except time is expressed in multiples of 2.4 microseconds.
	12	Same as TIME(2), except time is expressed in multiples of 2.4 microseconds.
	13	Same as TIME(3), except time is expressed in multiples of 2.4 microseconds.
	14	Returns the time since the last Halt/Load in 2.4 microseconds.
	15	Current date (in the format: 8"MMDDYY").
	16	Similar to TIME(6).
VALUE (mnemonic)	Mnemonic File Attribute	Integer value of the mnemonic value. (Refer to B6800 I/O Subsystems Reference Manual, form 5000185.)

BOOLEAN FUNCTION DESIGNATOR

Syntax

<Boolean function designator> ::= *<Boolean function identifier>* *<actual parameter part>*

<Boolean function identifier> ::= *<procedure identifier>* |
<Boolean intrinsic name>

<Boolean intrinsic name> ::= {a name of a Boolean intrinsic "known" by the **ALGOL** compiler}

Boolean Intrinsic Names

The following Boolean functions are intrinsic to **ALGOL**.

FUNCTION	PARAMETER(S)	RESULT
AVAILABLE(*<event designator>*)		Returns TRUE if the event is available and FALSE if not.
BOOLEAN(*<ae>*)		Returns the value of *<ae>* as a Boolean. If *<ae>* is DOUBLE then *<ae>* is truncated first.
HAPPENED(*<event designator>*)		Returns TRUE if the event has happened and FALSE if not.
OVERFLOW		Boolean. Returns the value of the OVERFLOW flip-flop.
READLOCK (*<be>*, *<Boolean variable>*)		Boolean. Taking only one memory cycle, *<be>* is stored into the *<Boolean variable>* and the prior content of the *<Boolean variable>* is returned.
TOGGLE		Boolean. Returns the value of the TRUE-FALSE flip-flop. It is SET or RESET by the scan and replace statements.

POINTER FUNCTION DESIGNATOR

Syntax

<pointer function designator> ::= **POINTER** (*<pointer parameters>*) |
READLOCK (*<pointer expression>* , *<pointer variable>*)

<pointer parameters> ::= *<array part>* |
<array part> , *<character size>* |
<array part> , *<pointer primary>*

<character size> ::= 4 | 6 | 8
<array part> ::= *<array row>* |
<subscripted variable>

Pointer Intrinsic Names

The following pointer function is intrinsic to **ALGOL.**

FUNCTION	PARAMETER(S)	RESULT
READLOCK (*<pe>*, *<pointer variable>*)		Taking only one memory cycle, *<pe>* is stored into the *<pointer variable>* and the prior content of the *<pointer variable>* is returned.

POINTER EXPRESSION

Syntax

<pointer expression> ::= <conditional pointer expression> |
<simple pointer expression>

<conditional pointer expression> ::= <if clause> <pointer expression> ELSE <pointer expression>

<simple pointer expression> ::= <pointer primary> <skip> |
<pointer assignment> |
<character array part>

<pointer primary> ::= <pointer identifier> |
(<pointer expression>) |
<case head> (<pointer expression list>) |
<pointer function designator>

<skip> ::= <empty> |
<adding operator> <primary> |
<adding operator> <arithmetic variable> := <arithmetic expression>

<pointer expression list> ::= <pointer expression> |
<pointer expression list> , <pointer expression>

<character array part> ::= <character array name> |
<character array row> |
<subscripted character array variable>

<character array name> ::= <character array identifier> |
<direct character array identifier>

<character array identifier> ::= <identifier>

<direct character array identifier> ::= <identifier>

<character array row> ::= <character array name> [<row designator>]

<subscripted character array variable> ::= <character array name>
[<subscript list>]

Examples

```
PTR
PTS+15
PTEMP + (X*Y) % NOTE THE NEED FOR PARENS HERE
PSORCE – INDX := X * Y % BUT NOT HERE
ARAY % WORD ARRAY
INXARAY[N]
HEXARAY % CHARACTER ARRAY
CASE VAL OF (PTR,PTS,PTEMP,PSORCE)
POINTER(INFO,6)
READLOCK(PTR,POLD)
```

Semantics

A *<pointer expression>* refers to a character position in an *<array row>*. Actually, it is an indexed or unindexed string descriptor, depending upon whether or not it points to the beginning of an *<array row>*. *<pointer expression>s* can take various forms in accordance with the primaries of which they are composed. Identifiers used as pointer primaries must be declared in a *<pointer declaration>*. If a *<pointer expression>* is enclosed in parentheses, it is evaluated first and its value is used as a *<pointer primary>*.

If the *<array part>* is of the form *<array name>*, the *<array name>* must reference a one-dimensional array.

POINTER INITIALIZATION

A pointer can be initialized either by a *<pointer assignment>* or by appearing as an *<update pointer>* in a *<scan statement>*, *<replace statement>*, or *<string relation>*.

POINTER ADJUSTMENT

If the *<skip>* construct is not *<empty>*, the pointer is adjusted by L characters to the right or left, where L is the *absolute value* of the *<arithmetic expression>*. If the *<adding operator>* is +, skipping is to the right; if it is –, skipping is to the left. Skipping to the right is defined to be incrementing the value of the character index and skipping to left as decrementing it.

NOTE

> A *<skip>* of the form +L, where L is less than or equal to zero, causes no adjustment of the pointer.

<pointer parameters>

The *<array part>* of the *<pointer parameters>* construct can be an *<array row>*, in which case the pointer references the beginning of the indicated row; or an *<array identifier>*, in which case the pointer references the first word of the array, which must be one-dimensional; or a *<subscripted variable>*, in which case the pointer references the indicated word. The *<character size>* part indicates the length, in bits, of the characters in the referenced array. If the *<character size>* does not appear, it is assumed to be the default value.

The *<array part>*, *<pointer primary>* alternative allows the use of another pointer to designate the size of a newly initialized pointer. For example,

```
POINTER P,Q; ARRAY A[0:5]; REAL R;
P:=POINTER(A,Q); P:=POINTER(A,CASE R OF (Q,P));
```

Pragmatics

The use of a *<pointer expression>* to skip up and down an array for more than a few words is expensive. To guard against the user "wandering out" of an array into core that is assigned to other arrays, each word of the source and/or destination is accessed in order to ensure that no memory protected words (at either end of the array) are encountered. For pointer moves greater than 18 bytes, it is much better to re-index from the base of the array by use of the **POINTER** (*<pointer parameters>*) intrinsic for word arrays or simply re-index the descriptor for character arrays.

APPENDIX A. RESERVED WORDS

RESERVED WORDS

All reserved words in B 7000/B 6000 Extended **ALGOL** have the syntactical structure of *<identifier>s*. The reserved words are divided into three types: type 1, type 2, and type 3.

Type 1 reserved words are those words that cannot be used as identifiers, that is, they cannot be associated with any entity, declared or specified, in the *<program>*. In the reserved word list, type 1 reserved words are denoted by (1). For example, **DIRECT** (1).

Type 2 reserved words are those words that can be declared to be *<identifier>s* (overriding their previous meaning). That is, everywhere within the scope of the declared or specified entity, the type 2 reserved word references the declared or specified entity and not the function normally referenced by the reserved word. In the reserved word list, type 2 reserved words are denoted by (2). For example, **FORMAL** (2).

Type 3 reserved words are those words that can be declared to be identifiers, but, where used in the language as specified by the syntax, have the reserved meaning. They are therefore "context sensitive". In other words, whenever an *<identifier>* that coincidentally spells a reserved word of type 3 is used in the language where the syntax calls for a reserved word of type 3, the *<identifier>* is not considered by the compiler to be a reference to some entity, but rather the reserved word of type 3. If, however, the *<identifier>* appears in the language where the syntax does not call for a reserved word of type 3, the *<identifier>* is taken by the compiler to be a reference to some entity declared or specified in the *<program>*, in which case the particular entity being referenced is determined by the rules of scope. Note the difference in the following example:

```
BEGIN
FILE F;
REAL KIND;
% IN THE NEXT STATEMENT, "KIND" REFERENCES THE REAL VARIABLE
KIND := 4.5;
% IN THE NEXT STATEMENT, "KIND" IS A TYPE THREE RESERVED WORD
FILE.KIND := VALUE (PRINTER);
WRITE (F, <"KIND =", R5.2>, KIND);
END.
```

Reserved words of type 3 are file mnemonics, file attributes, or task attributes. In the reserved word list, type 3 reserved words are denoted by (3,T) or (3,F), where the T signifies a task attribute, and the F signifies either a file mnemonic or file attribute. For example, **EXCEPTIONTASK** (3,T).

RESERVED WORDS

ABS (2)
ACCEPT (2)
ALGOLCODE (3,F)
ALGOLSYMBOL (3,F)
ALPHA (1)
AND (2)
ARCCOS (2)
ARCSIN (2)
ARCTAN (2)
ARCTAN2 (2)
AREACLASS (3,F)
AREAS (3,F)
AREASIZE (3,F)
ARRAY (1)
ASCII (1)
ASCIITOBCL (2)
ASCIITOEBCDIC (2)
ASCIITOHEX (2)
ASSIGNTIME (3,F)
ATANH (2)
ATEND (3,F)
ATTACH (2)
ATTERR (3,F)
ATTRIBUTERR (3,F)
ATTVALUE (3,F)
ATTYPE (3,F)
AVAILABLE (2)

BACKUP (3,F)
BACKUPDISK (3,F)
BACKUPPREFIX (3,T)
BASICCODE (3,F)
BASICSYMBOL (3,F)
BCL (1)
BCLTOASCII (2)
BCLTOEBCDIC (2)
BCLTOHEX (2)
BEGIN (1)
BINDERSYMBOL (3,F)
BLOCK (3,F)
BLOCKSIZE (3,F)
BOOLEAN (1)
BOUNDCODE (3,F)
BREAKHERE (3,F)
BREAKPOINT (2)
BUFFERS (3,F)
BY (2)

CALL (2)
CARRIAGECONTROL (3,F)
CASE (2)
CAUSE (2)
CAUSEANDRESET (2)
CDATA (3,F)
CENSUS (3,F)
CHARACTERS (3,F)
CHECKPOINT (2)
CLASS (3,T)
CLASSA (3,F)
CLASSB (3,F)
CLASSC (3,F)
CLOSE (2)
CLOSED (3,F)
COBOLCODE (3,F)
COBOLSYMBOL (3,F)
CODEFILE (3,F)
COMMENT (1)
COMPILERCODEFILE (3,F)
COMPILETIME (2)
CONTINUE (1)
CONTROLDECK (3,F)
COPIES (3,F)
COREESTIMATE (3,T)
COS (2)
COSH (2)
COTAN (2)
CRUNCHED (3,F)
CSEQDATA (3,F)
CURRENTBLOCK (3,F)
CYCLE (3,F)
CYLINDERMODE (3,F)

DABS (2)
DAND (2)
DARCCOS (2)
DARCSIN (2)
DARCTAN (2)
DARCTAN2 (2)
DATA (3,F)
DATAERROR (3,F)
DATE (3,F)
DCALGOLCODE (3,F)
DCALGOLSYMBOL (3,F)
DCOS (2)
DCOSH (2)
DEALLOCATE (2)
DECKGROUPNO (3,T)
DECLAREDPRIORITY (3,T)
DEFINE (2)
DELTA (2)
DENSITY (3,F)
DEQV (2)
DETACH (2)
DEXP (2)
DGAMMA (2)
DIGIIS (2)
DIGITS (2)
DIMP (2)
DINTEGER (2)
DIRECT (1)
DIRECTION (3,F)
DIRECTORY (3,F)
DIRECTORYCONTROL (2)
DISABLE (2)
DISK (3,F)
DISKHEADER (1)
DISKPACK (3,F)
DISPLAY (2)
DISPOSITION (3,F)
DIV (2)
DLGAMMA (2)
DLN (2)
DLOG (2)
DMAX (2)
DMIN (2)
DNABS (2)
DNOT (2)
DO (1)
DOUBLE (1)
DSCALELEFT (2)
DSQRT (2)
DTAN (2)
DTANH (2)
DUMP (2)
DUPLICATED (3,F)

EBCDIC (1)
EBCDICTOASCII (2)
EBCDICTOBCL (2)
EBCDICTOHEX (2)
ELAPSEDTIME (3,T)
ELSE (1)

ENABLE (2)
ENABLEINPUT (3,F)
END (1)
ENTER (2)
EOF (3,F)
EOFBITS (3,F)
EOFSEGMENT (3,F)
EQL (2)
EQV (2)
ERF (2)
ERFC (2)
ERRORTYPE (3,F)
ESPOLCODE (3,F)
ESPOLSYMBOL (3,F)
EUNUMBER (3,F)
EVENT (1)
EXCEPTIONEVENT (3,T)
EXCEPTIONTASK (3,T)
EXCLUSIVE (3,F)
EXTMODE (3,F)

FALSE (1)
FAMILY (3,F)
FAMILYSIZE (3,F)
FAST (3,F)
FILE (1)
FILECARDS (3,T)
FILEKIND (3,F)
FILETYPE (3,F)
FILL (2)
FIRSTONE (2)
FIRSTWORD (2)
FIX (2)
FLEXIBLE (3,F)
FOR (1)
FORMAL (2)
FORMAT (2)
FORMESSAGE (3,F)
FORMMESSAGE (3,F)
FORTRANCODE (3,F)
FORTRANSYMBOL (3,F)
FORWARD (2)
FREE (2)

GAMMA (2)
GEQ (2)
GO (1)
GTR (2)
GUARDFILE (3,F)

HAPPENED (2)
HEX (1)
HEXTOASCII (1)
HEXTOBCL (1)
HEXTOEBCDIC (1)
HIGH (3,F)
HISTORY (3,T)

IAD (3,F)
IF (1)
IMP (2)
IN (2)
INCREMENT (2)
INITIATOR (3,T)
INTEGER (1)
INTEGERT (2)
INTERCHANGE (3,F)
INTERRUPT (2)
INTMODE (3,F)
INTNAME (3,F)
INTRINSICINFO (2)
INTRINSICFILE (3,F)
IO (3,F)
IOADDRESS (2)
IOADJUST (2)
IOCANCEL (2)
IOCHARACTERS (2)
IOCLOCKS (3,F)
IOCOMPLETE (2)
IOCW (2)
IOEOF (2)
IOINERROR (3,F)
IOERRORTYPE (2)
IOMASK (2)
IOPENDING (2)
IORECORDNUM (2)
IORESULT (2)
IOTIME (2)
IOWORDS (2)
IS (2)
ISNT (2)

JOBNUMBER (3,F)
JOVIALCODE (3,F)
JOVIALSYMBOL (3,F)

KIND (3,F)

LABEL (1)
LABELTYPE (3,F)
LASTACCESSDATE (3,F)
LASTRECORD (3,F)
LASTSTATION (3,F)
LB (2)
LEQ (2)
LIBERATE (2)
LIBRARYCODE (3,F)
LINE (2)
LINENUM (3,F)
LIST (1)
LISTLOOKUP (2)
LN (2)
LNGAMMA (2)
LOCK (2)
LOCKED (3,T)
LOCKEDOUT (3,F)
LOG (2)
LONG (1)
LOW (3,F)
LSS (2)

MASKSEARCH (2)
MAX (2)
MAXCARDS (3,T)
MAXIOTIME (3,T)
MAXLINES (3,T)
MAXPROCTIME (3,T)
MAXRECSIZE (3,F)
MCPCODEFILE (3,F)
MEDIUM (3,F)
MEDIUMFAST (3,F)
MEDIUMSLOW (3,F)
MEMORYDUMP (2)
MERGE (2)
MESSAGE (1)
MIN (2)
MINRECSIZE (3,F)
MOD (2)
MODE (3,F)

RESERVED WORDS (Cont)

MONITOR (2)
MUX (2)
MYSELF (2)
MYUSE (3,F)

NABS (2)
NAME (3,T)
NEQ (2)
NEWUSER (3,F)
NO (2)
NOERROR (3,F)
NOINPUT (3,F)
NONSTANDARD (3,F)
NORMAL (3,F)
NORMALIZE (2)
NOT (2)
NUMBER (2)

OF (2)
OMITTED (3,F)
OMITTEDEOF (3,F)
ON (2)
ONES (2)
OPEN (3,F)
OPTION (3,T)
OPTIONAL (3,F)
OR (2)
ORGUNIT (3,T)
OTHERUSE (3,F)
OUT (3,F)
OVERFLOW (2)
OWN (1)

PACKHEADER (3,F)
PACKNAME (3,F)
PACKREBUILD (3,F)
PAGE (3,F)
PAGESIZE (3,F)
PAPER (3,F)
PAPERPUNCH (3,F)
PAPERREADER (3,F)
PARITY (3,F)
PARITYERROR (3,F)
PARTNER (3,T)
PETAPE (3,F)
PICTURE (2)
PLICODE (3,F)
PLISYMBOL (3,F)
POINTER (1)
POPULATION (3,F)
POTC (2)
POTL (2)
PRESENT (3,F)
PRINTER (3,F)
PRIVATE (3,F)
PROCEDURE (1)
PROCESS (2)
PROCESSIOTIME (3,T)
PROCESSTIME (3,T)
PROCURE (2)
PROGRAMDUMP (2)
PROPERTY (1)
PROTECTED (3,F)
PROTECTION (3,F)
PUNCH (3,F)
PURGE (2)

QUEUE (1)

RANDOM (2)
RB (2)
READ (2)
READCHECK (3,F)
READCHECKFAILURE (3,F)
READER (3,F)
READLOCK (2)
READPARITYERROR (3,F)
REAL (1)
RECEPTIONS (3,F)
RECONSTRUCTIONFILE (3,F)
RECORD (3,F)
RECORDINERROR (3,F)
RECORDKEY (3,F)
REEL (3,F)
REFERENCE (1)
REMOTE (3,F)
REPLACE (2)
RESET (2)
RESIDENT (3,F)
RESIZE (2)
RESTART (3,T)
REVERSE (3,F)
REWIND (2)
ROWADDRESS (3,F)
ROWCLASS (3,F)
ROWS (3,F)
ROWSINUSE (3,F)
ROWSIZE (3,F)
RUN (2)

SAVEFACTOR (3,F)
SAVE (3,F)
SCALELEFT (2)
SCALERIGHT (2)
SCALERIGHTF (2)
SCALERIGHTT (2)
SCAN (2)
SCREEN (3,F)
SDIGITS (2)
SECONDWORD (2)
SECURED (3,F)
SECURITYGUARD (3,F)
SECURITYTYPE (3,F)
SECURITYUSE (3,F)
SEEK (2)
SEQDATA (3,F)
SERIALNO (3,F)
SET (2)
SIGN (2)
SIN (2)
SINGLE (2)
SINGLEPACK (3,F)
SINH (2)
SIZE (2)
SIZE2 (3,F)
SIZEMODE (3,F)
SIZEOFFSET (3,F)
SKIP (2)
SLOW (3,F)
SORT (2)
SPACE (2)
SPEED (3,F)
SPO (3,F)
SQRT (2)
STACKER (2)
STACKHISTORY (3,T)
STACKNO (3,T)
STACKSIZE (3,T)
STANDARD (3,F)
STARTTIME (3,T)
STATE (3,F)

RESERVED WORDS (Cont)

STATION (2)
STATIONSDENIED (3,F)
STATUS (3,T)
STEP (1)
STOP (2)
STOPPOINT (3,T)
SUBSPACES (3,T)
SUNOTREADY (3,F)
SUPER (3,F)
SWAP (2)
SWITCH (1)
SYSTEMDIRECTORY (3,F)
SYSTEMDIRFILE (3,F)

TAN (2)
TANH (2)
TAPE (3,F)
TAPE7 (3,F)
TAPE9 (3,F)
TAPEREELRECORD (3,F)
TARGETTIME (3,T)
TASK (1)
TASKATTERR (3,T)
TASKFILE (3,T)
TASKVALUE (3,T)
TEMPORARY (3,F)
THEN (1)
THRU (2)
TIME (2)
TIMELIMIT (2)
TIMEOUT (3,F)
TIMES (2)
TITLE (3,F)
TO (2)
TOGGLE (2)
TRANSLATETABLE (1)
TRANSMISSIONO (3,F)
TRANSMISSIONS (3,F)
TRUE (1)
TRUTHSET (1)
TYPE (3,T)

UNITNO (3,F)
UNITS (3,F)
UNITSLEFT (3,F)
UNTIL (1)
UPDATED (3,F)
UPDATEFILE (3,F)
USEDATE (3,F)
USERCODE (3,T)
USERDATA (2)
USERDATALOCATOR (2)
USERDATAREBUILD (2)

VALUE (1)
VECTORMODE (2)
VERSION (3,F)
VERSIONDIRECTORY (3,F)

WAIT (2)
WAITANDRESET (2)
WHEN (2)
WHILE (1)
WIDTH (3,F)
WITH (2)
WORDS (2)
WRITE (2)

XALGOLCODE (3,F)
XALGOLSYMBOL (3,F)
XDISKFILE (3,F)
XFORTRANCODE (3,F)
XFORTRANSYMBOL (3,F)

ZIP (1)

APPENDIX B. PROGRAM CHARACTER AND WORD FORMATS

WORD NOTATION

The notation [m:n] is used in this manual to describe data word fields. The 48 accessible bits of a data word are considered to be numbered, with the left-most bit being bit 47 and the right-most bit being bit 0. In the notation used here, m denotes the number of the left-most bit of the field being described, and n denotes the number of bits in the field. For example, the word field (figure B–1) shown here (bits 28 through 24) would be described by [28:5]:

47	43	39	35	31	27	23	19	15	11	7	3
46	42	38	34	30	26	22	18	14	10	6	2
45	41	37	33	29	25	21	17	13	9	5	1
44	40	36	32	28	24	20	16	12	8	4	0

Figure B–1. Word Notation

Hexadecimal forms are used extensively in the manual to indicate word contents. Such forms are particularly suited to describing the value of a data word, since each digit in a hexadecimal form indicates the contents of a four-bit field. Such fields can be visualized as columns in the preceding picture of a data word.

SIGNS OF NUMERIC FIELDS

The sign of a numeric field is represented as follows:

8-bit characters — The sign is in the zone bits of the *least* significant character (bits 7 through 4 of the field). A bit configuration of 1101 (4"D") indicates a negative number; any other bit configuration indicates a positive number.

6-bit characters — The sign is in the zone bits of the *least* significant character (bits 5 and 4 of the field). A bit configuration of 10 indicates a negative number; any other bit configuration indicates a positive number.

4-bit digits — The sign is carried as a separate digit, and it is the *most* significant digit of the field. A bit configuration of 1101 (4"D") indicates a negative number; any other bit configuration indicates a positive number.

CHARACTER REPRESENTATION

Figures B–2 through B–4 illustrate the various character formats within a word.

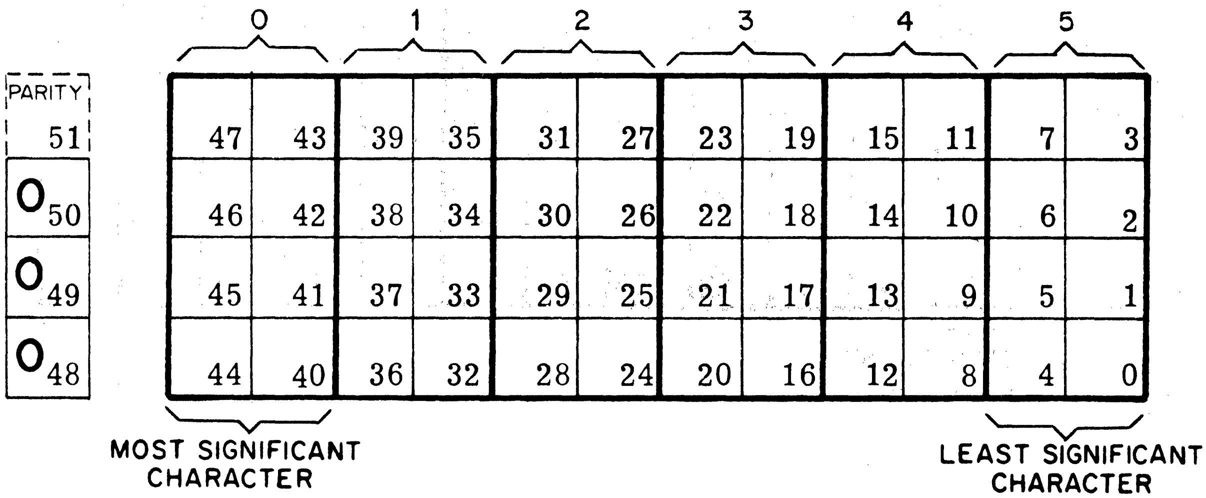


Figure B–2. 8-Bit Bytes (**EBCDIC** Code)

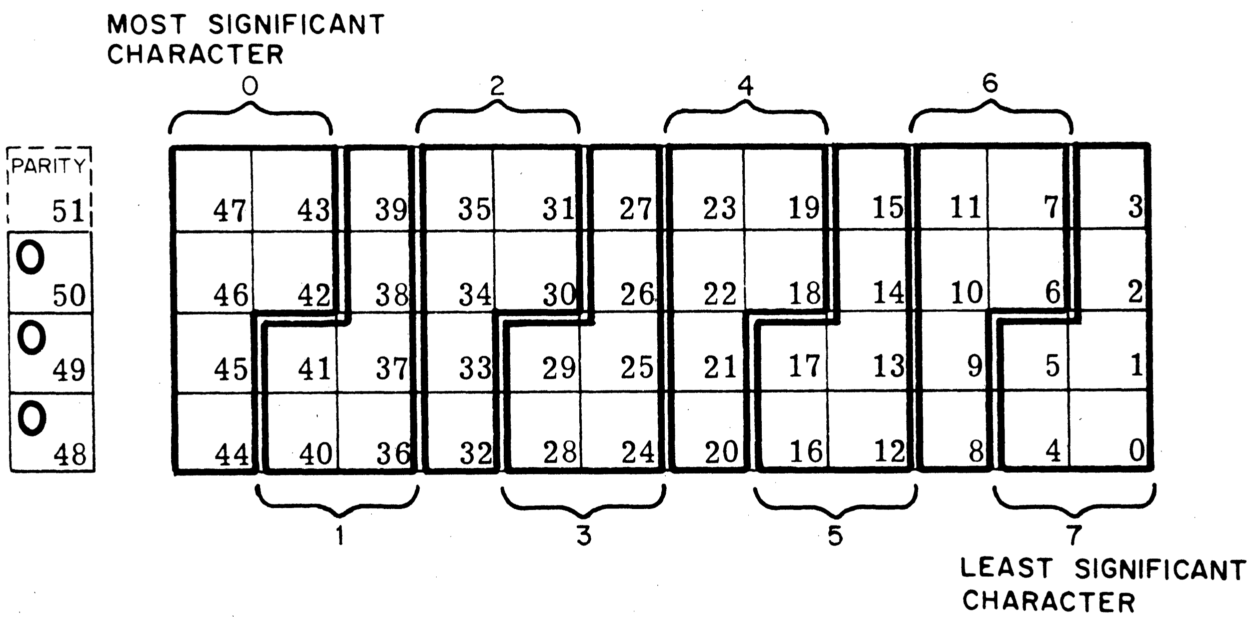


Figure B–3. 6-Bit Characters (**BCL** Code)

CHARACTER REPRESENTATION (Cont)

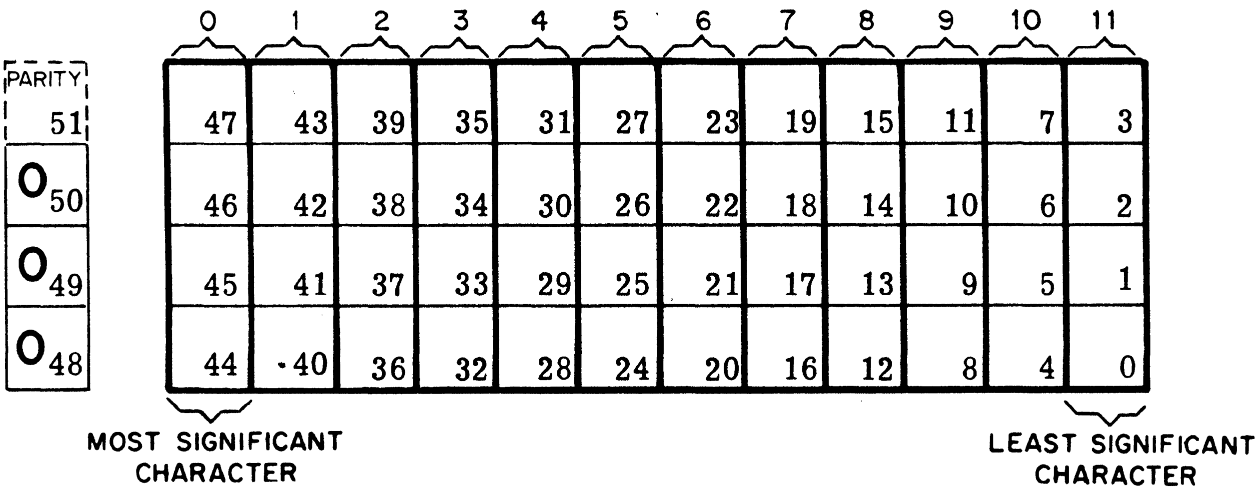
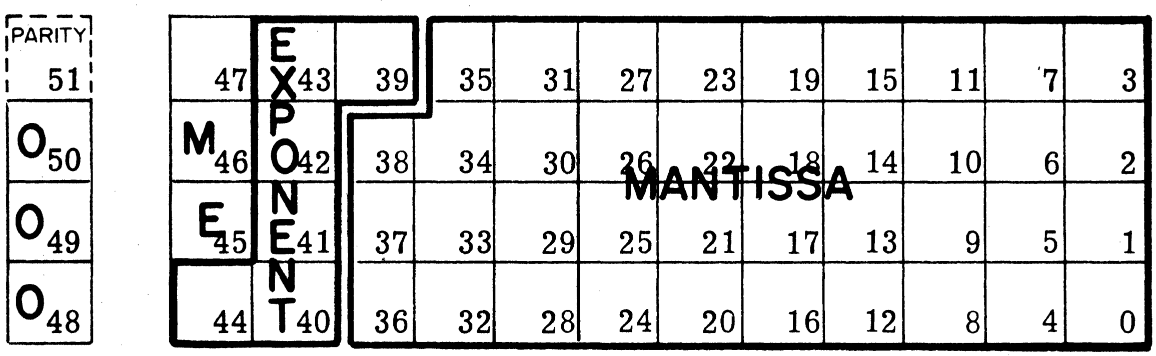


Figure B–4. 4-Bit Digits (Packed **BCD**)

SINGLE-PRECISION OPERAND

Real Variable

The real variable (figure B–5) requires *one* word of storage.



FIELD	USE
Tag	0
47	Not used
M	Sign of mantissa: 1 = negative, 0 = positive
E	Sign of exponent: 1 = negative, 0 = positive
44 => 39	Exponent
38 => 0	Mantissa

Figure B–5. Real Variable

A real variable requires *one* word of storage. Real (floating-point) values are represented internally in signed-magnitude mantissa-and-exponent notation. The sign of the mantissa is denoted by bit 46, and the sign of the exponent is denoted by bit 45 of the data word. A minus sign is denoted by a 1 in the appropriate bit. The magnitude of the exponent of the data item is contained in bits 44 through 39; hence, the exponent may not exceed a magnitude of 2**6 –1 (i.e., 63). The magnitude of the mantissa of the data item is contained in bits 38 through 0. This latter word field is identical to the magnitude field of a data word containing an integer value. Thus, an integer data word may be considered to be a real data word with an exponent field containing all zeros.

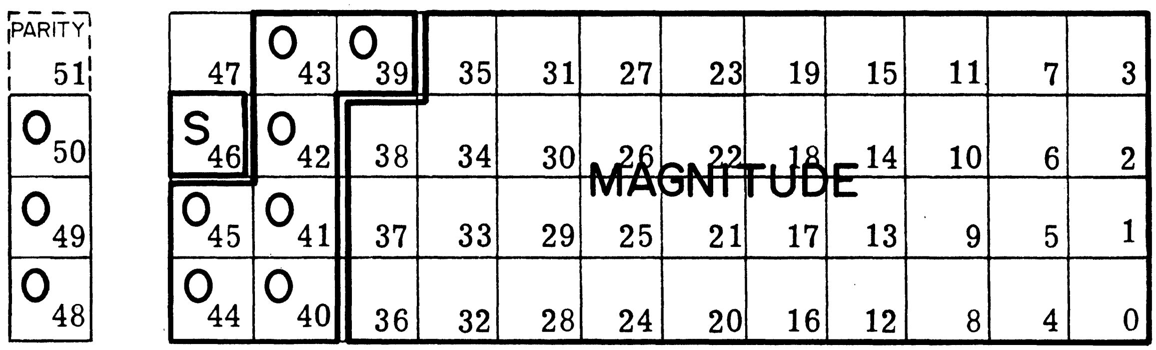
The value represented by a real data word may be obtained by the formula:

(mantissa) * 8 ** (exponent)

The magnitude of the mantissa is stored, left-normalized, within its word field. Therefore, the internal representation of the real value 0.5 (or .5E0) in hexadecimal and octal form is 26C000000000 and 0115400000000000, respectively. The exponent in this example is -13 and the mantissa is 4*8**12.

INTEGER VARIABLE

An integer variable (figure B–6) requires *one* word of storage.



FIELD	USE
Tag	0
47	Not used
S	Sign bit (0 = positive, 1 = negative)
45 => 39	Must contain all zeros
38 => 0	Magnitude

Figure B–6. Integer Variable

Integer values are represented internally in signed-magnitude notation. The sign of the value is denoted by bit 46 of the data word involved. This bit is 0 for positive values and 1 for negative values. The magnitude of the value is stored, right-justified in bits 38 through 0 and is preceded by zeros.

For example, the internal representation of the integer 10 described in hexadecimal form is as follows:

00000000000A

The following is the internal representation of -10:

40000000000A

The internal representation of 99999999999 is as follows:

00174876E7FF

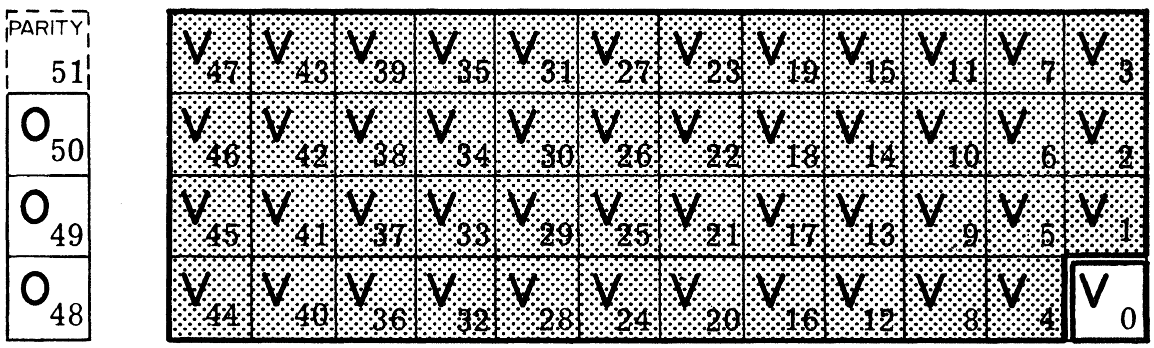
Finally, a hexadecimal form of:

007FFFFFFFFF

places into the data word the decimal quantity 549755813887, the maximum quantity an integer variable may possibly represent. A larger hexadecimal value would place a non-zero value in the [45:7] field of the data word, violating the internal format requirement of an integer value.

BOOLEAN VARIABLE

A Boolean variable (figure B–7) requires *one* word of storage.



FIELD	USE
Tag	0
47 => 0	A shaded V indicates a Boolean value can be used.
0	Boolean value for *<if clause>* syntax.

Figure B–7. Boolean Variable

Logical operations are performed on all 48 bits of the operand(s) involved, on a bit-by-bit basis.

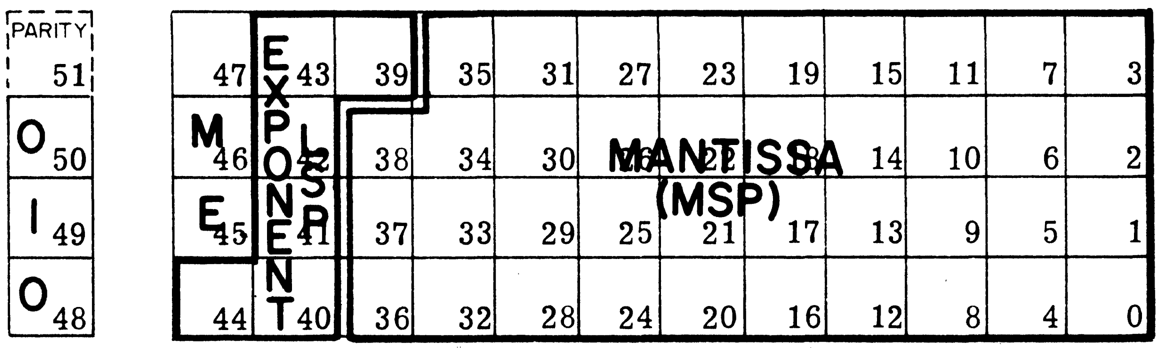
However, when using the *<if clause>* syntax, only the low-order bit (bit 0) is significant: 0 is **FALSE**, 1 is **TRUE.**

In the diagram above, the shaded V's represent the ability to use each bit of the entire 48-bit word as an individual Boolean value. This can be easily accomplished by means of *<partial word part>* and *<concatenation>* syntax.

DOUBLE-PRECISION OPERAND

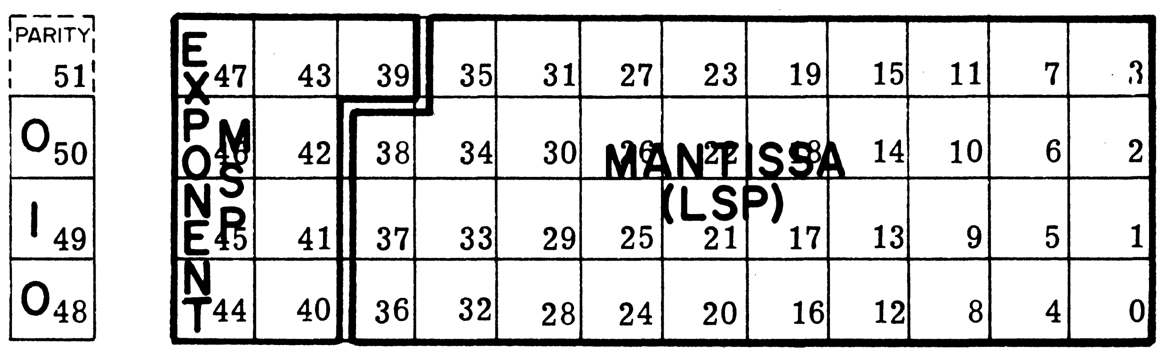
Double-Precision Variable

A double-precision variable (figures B–8 and B–9) requires *two* adjacent words of storage.



FIELD	USE
Tag	2
M	Sign of mantissa: 1 = negative, 0 = positive.
E	Sign of exponent: 1 = negative, 0 = positive.
44 => 39	Exponent, least-significant portion.
39 => 0	Mantissa, most-significant portion.

Figure B–8. First Word, Double-Precision Variable



FIELD	USE
Tag	2
47 => 39	Exponent, most-significant portion.
38 => 0	Mantissa, least-significant portion.

Figure B–9. Second Word, Double-Precision Variable

Double-precision values are represented internally in signed-magnitude mantissa-and-exponent notation, with a most-significant part and a least-significant part. The sign of the mantissa of the data item is contained in bit 46 of the first data word, and the sign of the exponent is contained in bit 45 of the first word. A minus sign is denoted by a 1 in the appropriate bit. The magnitude of the exponent of the data item is contained in a total of 15 bits. The first nine of these 15 bits (the most-significant part of the magnitude of the exponent) are contained in bits 47 through 39 of the second data word. The remaining six of these 15 bits (the least-significant part of the magnitude of the exponent) are contained in bits 44 through 39 of the first data word.

The magnitude of the mantissa of the data item is contained in a total of 78 bits which are divided in half, with each half being placed into one of the two data words. The first 39 bits (the most-significant part of the magnitude of the mantissa) are contained in bits 38 through 0 of the first data word. The remaining 39 bits (the least-significant part of the magnitude of the mantissa) are contained in bits 38 through 0 of the second data word. The mantissa sign bit applies to both portions of the mantissa.

The value represented by a double-precision data word pair may be obtained by the formula:

```
((most-significant part of mantissa)
+(least-significant part of mantissa)*8**(-13))
*8**((most-significant part of exponent)
    +(least-significant part of exponent)*2**6)
```

The magnitude of the mantissa is stored, left-normalized, within the total 78-bit field in which it is contained.

As an example, the internal two-word representation of the double-precision value 1D0 in hexadecimal form is:

```
261000000000
000000000000
```

The contents of the first of these two data words are identical to the contents of a data word representing the real value 1E0. When a value which does not exceed the limits of a real variable is assigned to a double-precision data word pair, then that data word pair can be represented in a form that will contain all zeros in the second data word; the first data word will contain a representation of the value identical to that which would have been formed if the value had been assigned to a real variable.

STRING DESCRIPTOR

The non-indexed and indexed string descriptors (figures B–10 and B–11) are described and illustrated as follows:

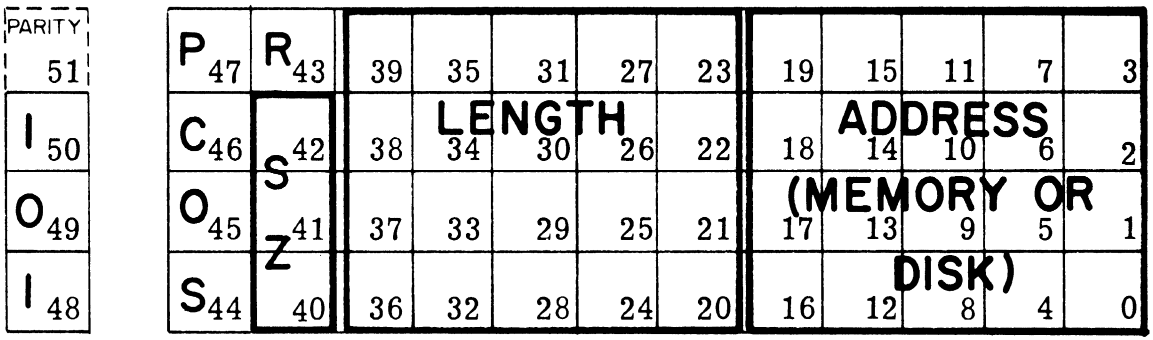


Figure B–10. String Descriptor (Non-Indexed)

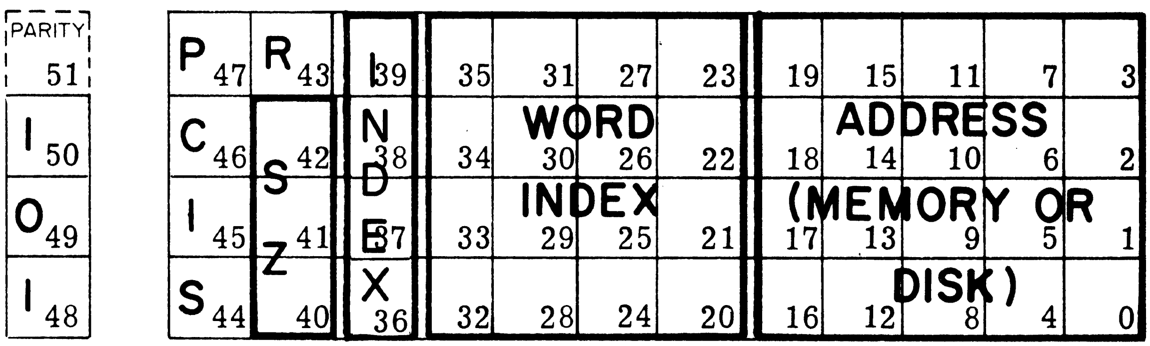


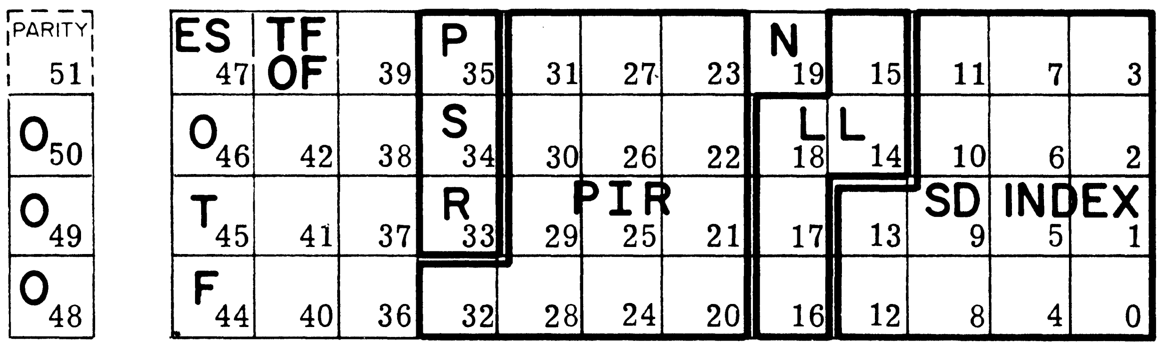
Figure B–11. String Descriptor (Indexed)

FIELD	USE
Tag	5
P	Presence bit: 1 = present, 0 = absent.
C	Copy bit: 1 = copy, 0 = **MOM** (original).
45	Index bit: 1 = indexing has taken place, 0 = indexing required.
S	Segmented bit: 1 = segmented data, 0 = non-segmented data.
R	Read-only bit: 1 = read only, 0 = read or write.
SZ	Character size: 100 = 8-bit, 011 = 6-bit, 010 = 4-bit.
39 => 20	Length of memory area (bit 45 = 0), or
39 => 36	Index (bit 45 = 1).
35 => 20	Word index (bit 45 = 1).
19 => 0	(bit 47 = 1) main memory address, or (bit 47 = 0) disk address.

The type transfer function **REAL** (*<pointer expression>*) returns, as a real value, the string descriptor with its tag set to zero. Depending on the point in time of accessing the string descriptor, the descriptor may or may not be "indexed", i.e., pointing beyond the very beginning of the *<array row>*. The typical use of the **REAL** (*<pointer expression>*) is to determine the amount of indexing that has taken place.

RETURN CONTROL WORD

The return control word (figure B–12) is described and illustrated as follows:



FIELD	USE
Tag	0
ES	External sign flip-flop.
O	Overflow flip-flop.
T	True/false flip-flop.
F	Float flip-flop.
TFOF	True/false occupied flip-flop.
PSR	Program syllable (0 through 5).
32 => 20	**PIR**: index to the program base register.
N	1 = control state, 0 = normal state.
LL	The lexicographical level of the calling procedure (at procedure entry).
13 => 0	**SD INDEX**: segment descriptor index; if bit 13 = 0, add value specified by bits 12:13 to D[0]; if bit 13 = 1, add value specified by bits 12:13 to D[1].

Figure B–12. Return Control Word

When the *<arithmetic-valued task attribute>* **STOPPOINT** is retrieved, the Return Control Word (with **TAG=O)** is returned. Note that the program associated with the task variable must in fact be *stopped* (either suspended or terminated); otherwise, the returned value is zero.

APPENDIX C. CHARACTER SETS AND CODING FORM

CHARACTER SET

The character set for the **ALGOL** language consists of the 256 characters in the Extended Binary Coded Decimal Interchange Code (**EBCDIC**) character set.

An **ALGOL** source program is represented by a set of ordered external records. The external records could come from tape, disk, card, remote device, etc. The character set of these external records may be **EBCDIC**, **ASCII**, or **BCL**. The external records from a remote device are often represented in the **ASCII** character set. If the external records are cards, these cards could have been punched in **BCL** or **EBCDIC**. However, regardless of these variations in the source external records, at compile-time, they are transformed into **EBCDIC** internal records by the B 7000/B 6000 System.

All subsequent remarks in this manual about the **ALGOL** character set pertain to the contents of these internal source records.

Default Character Size

Although the **ALGOL** source language is constructed only from the **EBCDIC** character set (whose character size is 8-bits), resulting object programs can work with character data whose character size is 6-bit (the **BCL** character set). In such object programs, when an assumption must be made concerning the character size (because the size was not explicitly supplied in the source language), the default size is 8-bit unless a **BCL** compiler option card is used, in which case the default size is 6-bit. (Refer to appendix D.)

CODING FORM

To facilitate keypunching, as well as to provide the programmer with a suggested format to follow in writing his program, printed programming forms are often used. An example of such a form appears in figure C–1.

It might be noted at this point that the compiler does not consider the information provided to it as a series of disjoint cards. Rather it treats the information as a continuous string of characters starting with the first column of a card, ending with column 72, and followed immediately by column 1 of the next card.

DATA REPRESENTATION

EBCDIC GRAPHIC	BCL GRAPHIC	HEX.	EBCDIC INTERNAL	DECIMAL VALUE	EBCDIC CARD CODE	OCTAL	BCL INTERNAL	BCL EXTERNAL	BCL CARD CODE	USASCII X3.4-1967
Blank		40	0100 0000	64	No Punches	60	11 0000	01 0000	No Punches	010 0000
[		4A	0100 1010	74	12 8 2	33	01 1011	11 1100	12 8 4	101 1011
.		4B	0100 1011	75	12 8 3	32	01 1010	11 1011	12 8 3	010 1110
<		4C	0100 1100	76	12 8 4	36	01 1110	11 1110	12 8 6	011 1100
(		4D	0100 1101	77	12 8 5	35	01 1101	11 1101	12 8 5	010 1000
+		4E	0100 1110	78	12 8 6			11 1010		010 1011
|	←	4F	0100 1111	79	12 8 7	37	01 1111	11 1111	12 8 7	111 1100
&		50	0101 0000	80	12	34	01 1100	11 0000	12	010 0110
]		5A	0101 1010	90	11 8 2	76	11 1110	01 1110	0 8 6	101 1101
$		5B	0101 1011	91	11 8 3	52	10 1010	10 1011	11 8 3	010 0100
*		5C	0101 1100	92	11 8 4	53	10 1011	10 1100	11 8 4	010 1010
)		5D	0101 1101	93	11 8 5	55	10 1101	10 1101	11 8 5	010 1001
;		5E	0101 1110	94	11 8 6	56	10 1110	10 1110	11 8 6	011 1011
¬	≤	5F	0101 1111	95	11 8 7	57	10 1111	10 1111	11 8 7	
-		60	0110 0000	96	11	54	10 1100	10 0000	11	101 1111
/		61	0110 0001	97	0 1	61	11 0001	01 0001	0 1	010 1111
,		6B	0110 1011	107	0 8 3	72	11 1010	01 1011	0 8 3	010 1100
%		6C	0110 1100	108	0 8 4	73	11 1011	01 1100	0 8 4	010 0101
_	≠	6D	0110 1101	109	0 8 5	74	11 1100	01 1010	0 8 2	010 1101
>		6E	0110 1110	110	0 8 6	16	00 1110	00 1110	8 6	011 1110
?		6F	0110 1111	111	0 8 7	14	00 1100	00 0000	*	011 1111
:		7A	0111 1010	122	8 2	15	00 1101	00 1101	8 5	011 1010
#		7B	0111 1011	123	8 3	12	00 1010	00 1011	8 3	010 0011
@		7C	0111 1100	124	8 4	13	00 1011	00 1100	8 4	100 0000
'	≥	7D	0111 1101	125	8 5	17	00 1111	00 1111	8 7	010 0111
=		7E	0111 1110	126	8 6	75	11 1101	01 1101	0 8 5	011 1101
"		7F	0111 1111	127	8 7	77	11 1111	01 1111	0 8 7	010 0010

EBCDIC GRAPHIC	BCL GRAPHIC	HEX.	EBCDIC INTERNAL	DECIMAL VALUE	EBCDIC CARD CODE	OCTAL	BCL INTERNAL	BCL EXTERNAL	BCL CARD CODE	USASCII X3.4-1967
(+)PZ	+	C0	1100 0000	192	12 0	20	01 0000	11 1010	12 0	
A		C1	1100 0001	193	12 1	21	01 0001	11 0001	12 1	100 0001
B		C2	1100 0010	194	12 2	22	01 0010	11 0010	12 2	100 0010
C		C3	1100 0011	195	12 3	23	01 0011	11 0011	12 3	100 0011
D		C4	1100 0100	196	12 4	24	01 0100	11 0100	12 4	100 0100
E		C5	1100 0101	197	12 5	25	01 0101	11 0101	12 5	100 0101
F		C6	1100 0110	198	12 6	26	01 0110	11 0110	12 6	100 0110
G		C7	1100 0111	199	12 7	27	01 0111	11 0111	12 7	100 0111
H		C8	1100 1000	200	12 8	30	01 1000	11 1000	12 8	100 1000
I		C9	1100 1001	201	12 9	31	01 1001	11 1001	12 9	100 1001
(!)MZ	x	D0	1101 0000	208	11 0	40	10 0000	10 1010	11 0	010 0001
J		D1	1101 0001	209	11 1	41	10 0001	10 0001	11 1	100 1010
K		D2	1101 0010	210	11 2	42	10 0010	10 0010	11 2	100 1011
L		D3	1101 0011	211	11 3	43	10 0011	10 0011	11 3	100 1100
M		D4	1101 0100	212	11 4	44	10 0100	10 0100	11 4	100 1101
N		D5	1101 0101	213	11 5	45	10 0101	10 0101	11 5	100 1110
O		D6	1101 0110	214	11 6	46	10 0110	10 0110	11 6	100 1111
P		D7	1101 0111	215	11 7	47	10 0111	10 0111	11 7	101 0000
Q		D8	1101 1000	216	11 8	50	10 1000	10 1000	11 8	101 0001
R		D9	1101 1001	217	11 9	51	10 1001	10 1001	11 9	101 0010
¢		E0	1110 0000	224	0 8 2			00 0000		
S		E2	1110 0010	226	0 2	62	11 0010	01 0010	0 2	101 0011
T		E3	1110 0011	227	0 3	63	11 0011	01 0011	0 3	101 0100
U		E4	1110 0100	228	0 4	64	11 0100	01 0100	0 4	101 0101
V		E5	1110 0101	229	0 5	65	11 0101	01 0101	0 5	101 0110
W		E6	1110 0110	230	0 6	66	11 0110	01 0110	0 6	101 0111
X		E7	1110 0111	231	0 7	67	11 0111	01 0111	0 7	101 1000
Y		E8	1110 1000	232	0 8	70	11 1000	01 1000	0 8	101 1001
Z		E9	1110 1001	233	0 9	71	11 1001	01 1001	0 9	101 1010
0		F0	1111 0000	240	0	00	00 0000	00 1010	0	011 0000
1		F1	1111 0001	241	1	01	00 0001	00 0001	1	011 0001
2		F2	1111 0010	242	2	02	00 0010	00 0010	2	011 0010
3		F3	1111 0011	243	3	03	00 0011	00 0100	3	011 0100
4		F4	1111 0100	244	4	04	00 0100	00 0100	4	011 0100

EBCDIC GRAPHIC	BCL GRAPHIC	HEX.	EBCDIC INTERNAL	DECIMAL VALUE	EBCDIC CARD CODE	OCTAL	BCL INTERNAL	BCL EXTERNAL	BCL CARD CODE	USASCII X3.4-1967
5		F5	1111 0101	245	5	05	00 0101	00 0101	5	011 0101
6		F6	1111 0110	246	6	06	00 0110	00 0110	6	011 0110
7		F7	1111 0111	247	7	07	00 0111	00 0111	7	011 0111
8		F8	1111 1000	248	8	10	00 1000	00 1000	8	011 1000
9		F9	1111 1001	249	9	11	00 1001	00 1001	9	011 1001

NOTES

1. **EBCDIC** 0100 1110 also translates to **BCL** 11 1010.

2. **EBCDIC** 1100 1111 is translated to **BCL** 00 0000 with an additional flag bit on the most-significant bit line (8th bit). This function is used by the unbuffered printer to stop scanning.

3. **EBCDIC** 1110 0000 is translated to **BCL** 00 0000 with an additional flag bit on the next to most significant bit line (8th bit). As the print drums have 64 graphics and spaces, this signal can be used to print the 64th graphic. The 64th graphic is a **"CR"** for **BCL** drums and a **"¢"** for **EBCDIC** drums.

4. The remaining 189 **EBCDIC** codes are translated to **BCL** 00 0000 (?code).

5. The **EBCDIC** graphics and **BCL** graphics are the same except as follows:

BCL	EBCDIC
≥	' (single quote)
x (multiply)	!
≤	¬ (not)
≠	_ (underscore)
←	| (or)

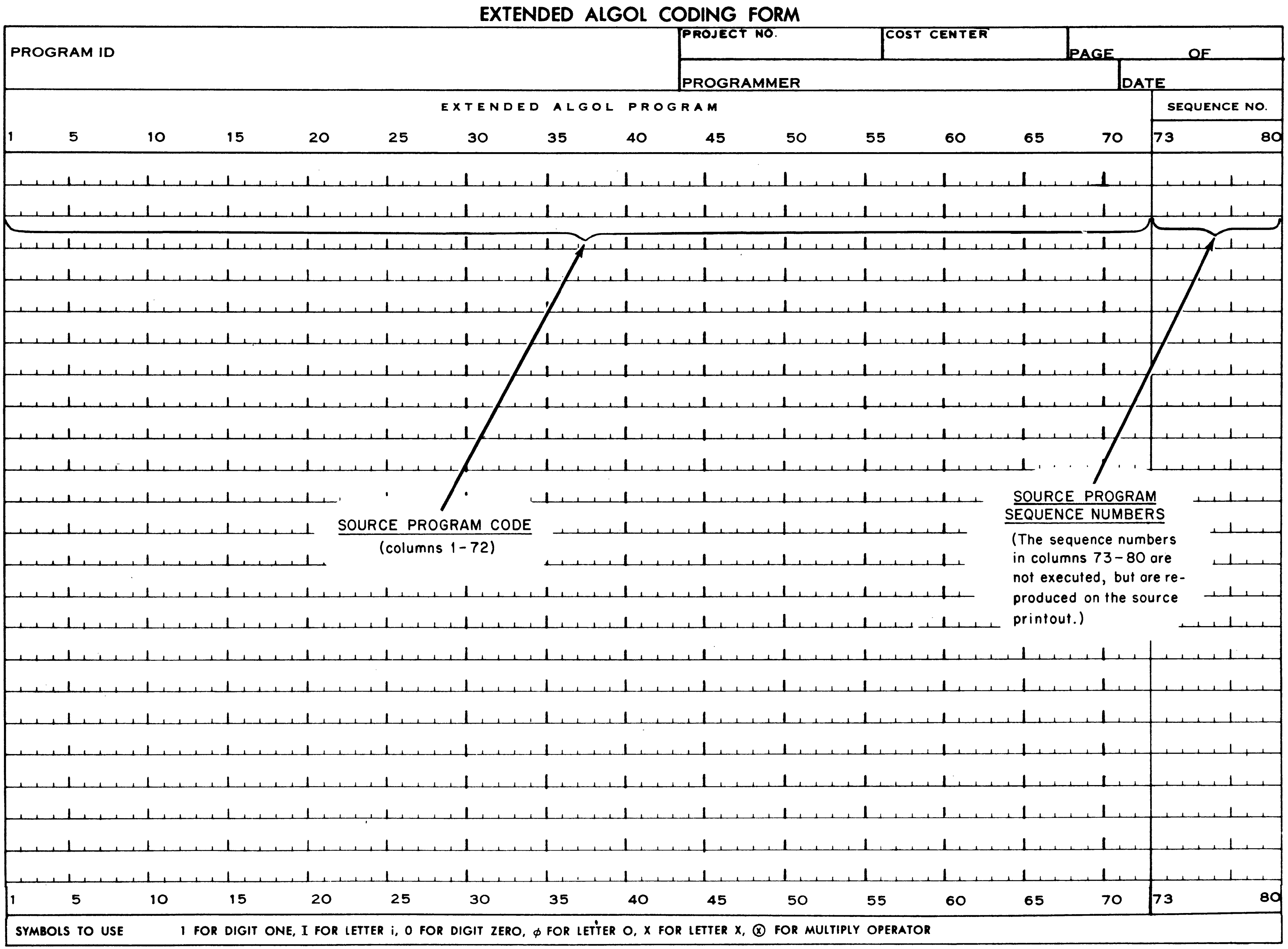


Figure C–1. Extended **ALGOL** Coding Form

FORM 1038452

APPENDIX D. COMPILE-TIME OPTIONS

COMPILER CONTROL STATEMENTS

The user is provided with the compile-time ability to control the manner in which the compiler processes the source input that it accepts. The user can specify the manner in which the compiler is to receive the source input, the consequences of certain syntax errors, and the form of the generated compiler output. The compiler control statement is the medium by which these constraints are communicated to the compiler. Such statements are entered into the compiler by cards in the same manner as source language statements. Compiler control statements, entered as input to the compiler via option control cards, can occur at any point in the compiler input files and must contain only compiler control information.

Option control cards are recognized either unconditionally or when the compiler is looking for the next syntactic item; the difference in the treatment depends on the column where the $ sign is found.

Option control cards with a $ sign in either column 1 or 2 (in the latter case with a blank in column one) are unconditionally recognized and processed. Option control cards with a $ sign in columns 3 through 72 are recognized only when the compiler is expecting a new syntactic item. In particular, such an option control card is not recognized in at least the following instances:

a. following a %
b. while processing a format specification (an entire format set of phrases is treated as only one syntactic item)
c. within commentary
d. while OMITting
e. following the @ in a numeric constant

Columns 73 through 80 are reserved for an eight-digit sequence number. All blanks in columns 73 through 80 represent the lowest-value sequence number. An option control card with no other compiler information causes the card image in the secondary input file that has the same sequence number to be ignored.

The basic element of compiler control information is the compiler option, which can be invoked by the appearance of its name on an option control card. Two mutually exclusive states are associated with the majority of these options: **SET** and **RESET**; various compiler functions are dependent upon the states of such options. Default states are assigned to these compiler options, and the desired state of such an option can be specified on an option control card. Such option control cards can also contain arguments associated with the option. The balance of compiler options are parameter options with which no states are associated. The functions performed by these latter options are initiated by the appearance on an option control card of the appropriate option name and any related arguments.

Option control cards can contain the following information items in addition to the initial $ and the terminal sequence number:

a. Option actions that assign states to indicated standard options.
b. Option names and/or associated option arguments, that is, literals, etc., that are connected with the function of the options.

OPTION CONTROL CARDS

Syntax

<option control card> ::= $ *<option list>* *<option group list>*

<option list> ::= *<empty>* |
    <option list> *<option>*

<option group list> ::= *<empty>* |
    <option group list> *<option group>*

<option> ::= **AREACLASS | ASCII | AUTOBIND |**
BCL | BEGINSEGMENT | BIND | BINDER | BREAKHOST | BREAKPOINT | B7700TOG |
CHECK | CODE |
DOUBLESPACE | DUMPINFO |
ENDSEGMENT | ERRLIST | EXTERNAL |
FORMAT |
GO | GO TO |
HOST |
INCLNEW | INCLSEQ | INCLUDE | INITIALIZE | INSTALLATION | INTRINSICS |
LEVEL | LIBRARY | LINEINFO | LIST | LISTDELETED | LISTINCL | LISTOMITTED |
LISTP | LOADINFO |
MAKEHOST | MCP | MERGE |
NEW | NEWSEQERR | NOBINDINFO | NOSTACKARRAYS | NOXREFLIST |
OMIT | OPTIMIZE |
PAGE | PURGE |
SEGS | SEGDESCABOVE | SEPCOMP | SEQ | SEQERR | SINGLE | STACK | STATISTICS |
STOP |
TIME |
USE |
<user option> |
VERSION | VOID | VOIDT |
WRITEAFTER |
XDEC | XREF | XREFFILES | XREFS |
$

<option group> ::= *<option action>* *<option list>* *<parameter>* |
    <user option>

<option action> ::= **POP** |
RESET |
SET

<parameter> ::= *<error limit>* |
<sequence increment> |
<sequence base> |
<areaclass value> |
<outer level>

<error limit> ::= **LIMIT** *<unsigned integer>*

<sequence increment> ::= + *<unsigned integer>*

<sequence base> ::= *<unsigned integer>*

<outer level> ::= **LEVEL** *<unsigned integer>*

<areaclass value> ::= *<unsigned integer>*

<user option> ::= {word used for specific user option}

OPTION ACTIONS

A purpose of a compiler control statement could be the assignment of a desired state (**SET** or **RESET**) with an indicated compiler option(s). Such a control statement must begin with either an explicit or an implicit *<option action>*. An explicit *<option action>* is defined as one of the following mnemonics: **SET**, **RESET**, or **POP**.

An implicit *<option action>* is indicated when a compiler control statement contains only the names of options and no explicit *<option action>*. In the latter case, all options named in the compiler control statement are assigned the state **SET**, and all other options are assigned the state **RESET**.

If a compiler control statement begins with the *<option action>* **SET**, the options following the *<option action>* are assigned the state **SET**; the states of all other options are unchanged. If the compiler control statement begins with the *<option action>* **RESET**, the options following the *<option action>* are assigned the state **RESET**; the states of all other options are unchanged. If the specified *<option action>* is **POP**, then the options indicated revert to their immediate previous states; their states become **RESET** if these options have not been changed previously from their default states. The states of all other options are unchanged. The following statements are examples of compiler control statements employing the **SET**, **RESET**, and **POP** *<option action>s*.

```
$ SET LIST SINGLE INCLNEW
$ RESET VOID
$ POP NEW NEWSEQERR
$ SET SEQ 0+100
```

An option that has a default state of **RESET** is initially assigned a 48-bit stack word filled with zeros; an option that has a default state of **SET** is initially assigned a 48-bit stack word with a 1 on top and zeros in the remaining positions. The top stack position denotes the state of the option at any time. Each **SET** option action causes the stacks allocated to the designated standard options to be pushed down one bit and a 1 to be placed on the top of each of these stacks. Each **RESET** causes the appropriate option stacks to be pushed down one bit and a 0 to be placed on the tops of these stacks. **POP** causes the stacks corresponding to the designated options to be **POP**ped up one bit, causing the associated options to revert to their immediate previous states. Since the size of these option stacks is 48 bits, a maximum history of 48 states can be recorded. When an option control card appears that has a standard option name and an implicit option action, the resultant action is identical to that which would have resulted had all 48 bits of each standard option stack been **RESET** and followed by an explicit **SET** performed on each indicated option. For example, after the appearance of an option control card containing:

```
$ CODE
```

the history stack for the **CODE** option contains a 1 in the top stack psoition and all zeros in the following positions. The history stack for each of the other compiler options (**LIST**, **VOIDT**, etc.) would then contain all zeros. A compiler control statement that applies to compiler options begins with an explicit or implicit *<option action>* and contains a list of options to which the *<option action>* is to apply. This statement ends when the next implicit *<option action>* is encountered on the compiler control card or when a percent sign is encountered or column 72 of the card is reached. The compiler options affected by the compiler control card retain the indicated states for all input cards with sequence numbers greater than the sequence number on the compiler control card that has the control statement, or the physically succeeding input cards for a deck in which all sequence numbers are blank, until another compiler control card is encountered that alters the option states. A compiler control statement can also contain any parameter option name except **INCLUDE**, and the action initiated by the appearance of the option name still results. The following illustration (figure D–1) is an example of a card that has compiler control statements employing option actions:

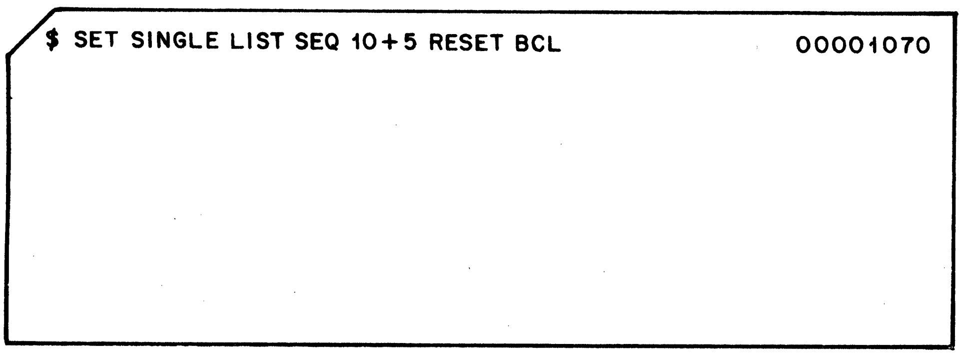


Figure D–1. Option Control Card

The option control card assigns the state **SET** to the options **SINGLE, LIST,** and **SEQ,** with the sequencing arguments of 10 and +5. It also assigns the state **RESET** to the option **BCL.** The card has the sequence number 00001070 in columns 73 through 80.

OPTIONS

The compiler recognizes the following identifiers as valid compiler option names:

AREACLASS	**GO**	**LOADINFO**	**SINGLE**
ASCII	**GO TO**	**MAKEHOST**	**STACK**
AUTOBIND	**HOST**	**MCP**	**STATISTICS**
BCL	**INCLNEW**	**MERGE**	**STOP**
BEGINSEGMENT	**INCLSEQ**	**NEW**	**TIME**
BIND	**INCLUDE**	**NEWSEQERR**	**USE**
BINDER	**INITIALIZE**	**NOBINDINFO**	<*user option*>
BREAKHOST	**INSTALLATION**	**NOSTACKARRAYS**	**VERSION**
BREAKPOINT	**INTRINSICS**	**NOXREFLIST**	**VOID**
B7700TOG	**LEVEL**	**OMIT**	**VOIDT**
CHECK	**LIBRARY**	**OPTIMIZE**	**WRITEAFTER**
CODE	**LIMIT**	**PAGE**	**XDECS**
DOUBLESPACE	**LINEINFO**	**PURGE**	**XREF**
DUMPINFO	**LIST**	**SEGDESCABOVE**	**XREFFILES**
ENDSEGMENT	**LISTDELETED**	**SEGS**	**XREFS**
ERRLIST	**LISTINCL**	**SEPCOMP**	**$**
EXTERNAL	**LISTOMITTED**	**SEQ**	
FORMAT	**LISTP**	**SEQERR**	

The appearance of a *<parameter>* option on an option control card with an implicit *<option action>* (no **SET**, **RESET**, or **POP**) does not result in the **RESET**ting of any options. The names of these parameter options are themselves compiler control statements and can appear on an option control card with other compiler control statements, *except* for the **INCLUDE** and **GO TO** options, which must appear alone on an option control card.

NOTE

The appearance of a one-to eight-digit unsigned integer or such an integer preceded by a + symbol constitutes a parameter control statement used as an argument associated with the **SEQ** option.

The appearance on an option control card of any option name that is not contained in the preceding list constitutes a compilation error, except for the *<user option>* option name.

The compiler options are discussed alphabetically in the following paragraphs. The default state of each *<option>* is indicated in parentheses following the *<option>* name; the function performed by the *<option>* is discussed in the paragraph accompanying the same.

The default state of the **LIST** option is **SET** and the default state of the balance of the standard options is **RESET** unless the compiler is employed by the **CANDE** Language. If the compiler is called from **CANDE**, the default state of the **ERRLIST** option is **SET** and the default state of the balance of the standard options is **RESET**.

<empty> has no effect on other *<option>s,* However, if there is a card image on the symbolic file with the same sequence number as the *<empty>* *<option control card>*, the image on the symbolic file is deleted.

The compiler options are as follows:

AREACLASS (RESET)

The **AREACLASS** option assigns the value specified in the *<areaclass value>* to the **AREACLASS** file attribute of the object code file when such a file is produced by the compiler.

ASCII

The **ASCII** option sets the default character size to 7-bit. Character arrays can be declared to be of type **ASCII**. Pointers become **ASCII** pointers by giving them a length attribute of seven (although each **ASCII** character still takes up eight bits).

ASCII may be used in the **TRUTHSET** and **TRANSLATETABLE** constructs to permit software comparisons and translations within the **ASCII** character set. However, because of hardware limitations, it is not permitted to replace an **ASCII** pointer for a specified number of digits. In addition, the integer and double type transfer functions for pointers are not available for **ASCII** pointers. These may result in errors at execution time.

AUTOBIND (RESET)

The **AUTOBIND** compiler option combines the processes of compiling and program binding into one job. During compilation, the compiler produces a set of instructions to be passed to the binder. In many cases, these binder instructions are self-sufficient for binding purposes and the user need not be concerned with binder control cards. In those cases where binder instructions are required, the user can insert binder control cards.

The **AUTOBIND** compiler option can be **SET** or **RESET** at any point throughout compilation. However, it is recommended that it be **SET** or **RESET** only once at the beginning of compilation for the following reasons:

a. Only the status of the **AUTOBIND** compiler option at the end of compilation is significant. Specifically, if four procedures are being compiled, the first three with the **AUTOBIND** option **RESET** and the last one with the **AUTOBIND** option **SET**, the binder still attempts to bind all four procedures to the specified host.

b. A compiler-and-go on a separate procedure with the **AUTOBIND** option **RESET** will not be executed. If the **AUTOBIND** option is **SET** throughout compilation, execution of the resultant program takes place after binding.

In **ALGOL**, a separate procedure compiled at level two or an outer block may serve as a "host" for binding. Separate **ALGOL** procedures compiled at level three (default level) may be bound into a host. At most, one host may be compiled in a job along with any number of separate procedures. The host must be the last program unit compiled. If an appropriate host file is compiled with **AUTOBIND SET**, it is assumed to be the host for binding. This assumption cannot be overridden by either of the methods given next for specifying a host. If no eligible host is being compiled, a host must be specified. Two methods are available:

<1> COMPILER FILE HOST (TITLE = FILEDI/..../FILEIDN)

or a BINDER host card, such as

$ HOST IS FILEDI/..../FILEIDN

The code file of any level three procedure compiled with **AUTOBIND SET** is marked as non-executable. If not executed via inter-program communication, the procedure must be bound into a host file by the binder before being executed.

Code files of any compiled level three procedures (or higher) are purged after being bound into a host by **AUTOBIND**. To retain such as a code file, it is necessary to refer to it specifically in a binder control statement. Either of the following statements will allow the procedure's code file to remain:

$ BIND PROCEDURENAME

or

$ EXTERNAL PROCEDURENAME

The first statement performs the same function as the default compiler-generated bind statement, except the code file will not be purged. The second statement instructs the binder not to bind the procedure into the host even though it has been compiled with **AUTOBIND SET** and there is an external reference to it in the host.

BCL (RESET)

The **BCL** option **SET**s the default character size of the object programs, pointers, strings and data to 6-bit.

BEGINSEGMENT (RESET)

The **BEGINSEGMENT** and **ENDSEGMENT** options allow user control of procedure and block segmentation. Procedures and blocks encountered between the **BEGINSEGMENT** and **ENDSEGMENT** options are placed in the same segment. The **BEGINSEGMENT** option must appear before the declaration of the first procedure or block to be included in the user segment. The **ENDSEGMENT** option must appear after the last source image of the last procedure or block in the user segment. The first procedure or block in the user segment must be the one that the compiler normally segments. Only procedures and blocks completely contained within a procedure or block in the segment can be included in a user segment. External procedures cannot be declared in a user segment.

User segments can be nested, that is, a **BEGINSEGMENT** can appear in a user segment. In this case an **ENDSEGMENT** option applies to the user segment currently being compiled.

If a **BEGINSEGMENT** option appears before the beginning of a separately compiled procedure, an **ENDSEGMENT** option control card is assumed at the end of the procedure, even if none appears. The driver procedure created for procedures compiled at lex-level 3 is always in a different segment.

The **BEGINSEGMENT** and **ENDSEGMENT** options can be **SET**, **RESET**, or **POP**ped. The printout for segment information is modified for user segments. User segments are numbered consecutively in a program, beginning with 1. That is, the first **BEGINSEGMENT** creates **USERSEGMENT1**. The second **BEGINSEGMENT** creates **USERSEGMENT2**. At the beginning of a user segment, its segment number is printed out. The length of each user segment is printed at its end. Procedures or blocks that are normally segmented, but are not because of user segmentation, print out as being "in" a particular segment.

Forward procedure declarations are not affected by user segmentation.

A procedure or block cannot be split across user segments.

If more than one **BEGINSEGMENT** option control card appears before a procedure, the warning message **EXTRA $BEGINSEGMENT IGNORED** is printed. If an **ENDSEGMENT** option control card appears when the user is not controlling segmentation, the warning message **EXTRA ENDSEGMENT IGNORED** is printed.

Another purpose of **BEGINSEGMENT** and **ENDSEGMENT** is to allow the adroit programmer the ability to segregate infrequently called procedures from frequently called procedures; that is, group frequently called procedures into one segment to reduce page faults of code segments.

BIND (autobinding only)

Format is similar to dollar sign ($) option, except that it is used to pass control statements to BINDER when autobinding.

BINDER (autobinding only)

Allows passing of compiler options when autobinding. The compiler "strips off" the word BINDER and passes the rest of the card intact as an option card to BINDER.

BREAKHOST (RESET)

This option must be set in the outer block of any program which uses **BREAKPOINT** in order to create the necessary environment for interactive debugging. A part of this environment is the creation of a remote file. Note that this option must be set after the first **BEGIN** of the program.

If a program to be debugged has a remote input file, then this option must be modified to allow the **BREAKPOINT** intrinsic to pick up the user's remote file as only one remote input file may be open per station. In this case, the syntax is **$SET BREAKHOST (INFILENAME)**, where **INFILENAME** is the name of the user's remote file. (Refer to the **BREAKPOINT** compiler option.)

BREAKPOINT

In the range of the **SET**, **POP** pair each **ALGOL** statement has a call on the **BREAKPOINT** intrinsic emitted after it. A user program's execution will stop (break) after each statement in this range to allow debugging via **BREAKPOINT** commands. (Refer to the **BREAKHOST** compiler option.)

EXAMPLE PROGRAM:

```
4000000   BEGIN
4000500   ARRAY D[0:5,0:8];
4001000   REAL R, S;
4002000   $SET BREAKHOST
4003000   BOOLEAN BOO;
4003100   EBCDIC ARRAY EB[0:100];
4004000   ARRAY A[0:11];
4005000   POINTER P4, P8;
4006000   PROCEDURE PROC;
4007000         BEGIN
4008000         REAL L;
4009000   $SET BREAKPOINT
4001000         L :=R;
4011000         S :=REAL (NOT FALSE);
4012000   $POP BREAKPOINT
4013000         END;
4015000   $SET BREAKPOINT
4016000   P8 :=POINTER(A);
4017000   P4 :=POINTER(A[5],4);
4018000   REPLACE P8 BY "ABCK" FOR 44;
4018100   REPLACE EB BY "ABCDEF";
4019000   D[0,6] :=474;
4020000   BOO:=TRUE;
4021000   R:=4" 1234567";
4022000   S:=66;
4023000   PROC;
4024000   END.
```

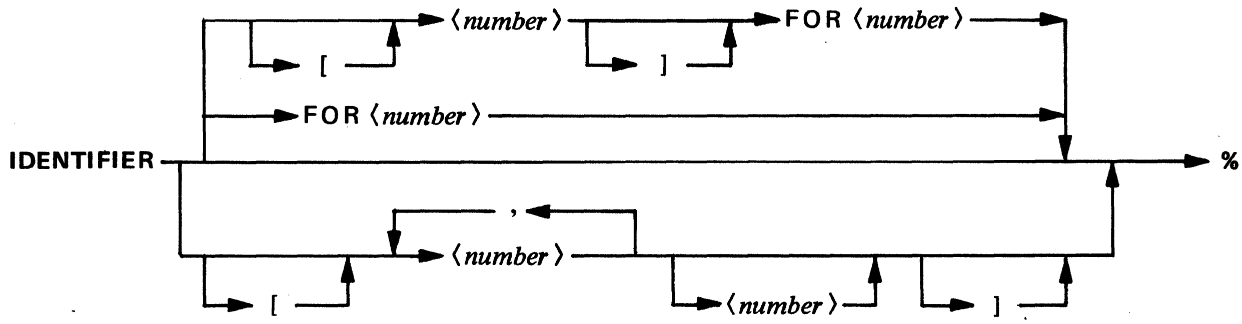
After a program is linked to the **BREAKPOINT** intrinsic, the program's execution will cause calls on the intrinsic after each statement in the range of a **$SET,POP BREAKPOINT** pair and at each explicit **BREAKPOINT** call. In the example program, execution will proceed normally until statement 4016000 where **BREAKPOINT** will be called. **BREAKPOINT** will thereafter be called after each statement and in procedure **PROC**.

BREAKPOINT INTRINSIC

Commands to the **BREAKPOINT INTRINSIC** are grouped into three modes: identifier mode, / mode, and & mode.

IDENTIFIER MODE

This mode allows access to values of variables in an **ALGOL** program. The syntax for this mode is:

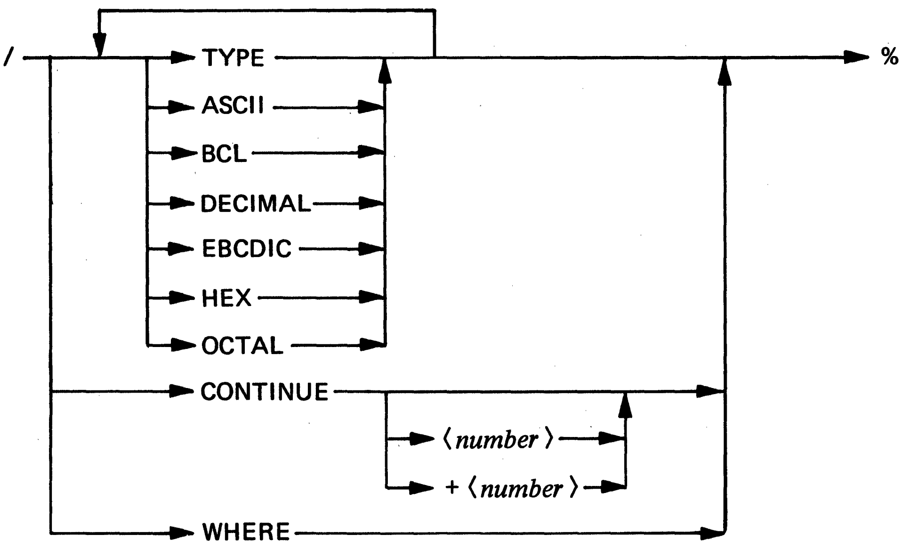


Example commands are:

R	%CAUSES BREAKPOINT TO RETURN THE %VALUE OF R
A[0-6]	% VALUE OF FIRST SIX ELEMENTS OF %ARRAY A
D[1,2]	% VALUE OF D[1,2]
P8 FOR 11	% RETURNS 11 CHARACTERS POINTED AT % BY P8

/ MODE

Commands in this mode must have a / as the first **CHARACTER** of the command line. These commands affect the present or future state of **BREAKPOINT** by changing the output mode of identifiers, indicating where in a user program execution has been stopped, or allowing a program's execution to continue. The syntax for this mode is:



Command names need not be typed to their full length for recognition.

FORMATTING CONTROL COMMANDS –

/BCL	**VARIABLE OUTPUT IN BCL**
/EBCDIC	**VARIABLE OUTPUT IN EBCDIC**
/DECIMAL	**VARIABLE OUTPUT IN DECIMAL**
/HEX	**VARIABLE OUTPUT IN HEX**
/OCTAL	**VARIABLE OUTPUT IN OCTAL**
/TYPE	**VARIABLE OUTPUT ACCORDING TO ITS DECLARATION IN THE USER'S PROGRAM. THIS IS THE DEFAULT OUTPUT MODE.**

Each new / line of a formatting output command causes the previous formats to be reset and the commands following the / to be the output types until new / formats are input.

EXECUTION STATE OUTPUT

/WHERE

Outputs the sequence number where the program's execution is stopped.

CONTINUATION OF PROGRAM EXECUTION

In the range of a **$SET, POP BREAKPOINT** pair it is often desired that **BREAKPOINT** action not be taken after every statement. To provide this capability, two forms of skip commands are provided.

/CONTINUE<*seqno*>

This causes the user program to execute until <*seqno*> is reached.

/CONTINUE+<*skipno*>

This causes <*skipno*> statements to be skipped. This command is useful when there exists more than one statement on a line.

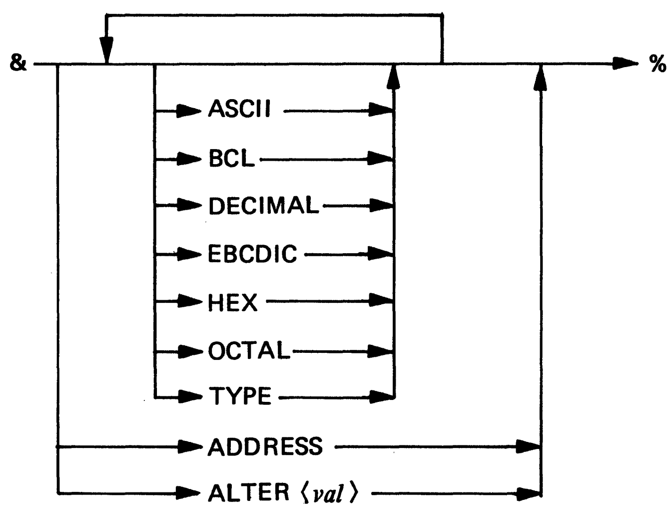
/CONTINUE and /CONTINUE+1 are equivalent.

Example commands:

/EBCDIC H DECIMAL
/C 73201000

& MODE

Commands in this mode must have an & as the first character of the command line. $ mode commands affect the last identifier that was examined in identifier mode. These commands allow the last identifier to be output in different formats, its address couple to be observed, and its value to be modified.



Command names need not be typed to their full length for recognition.

FORMATTING CONTROL

&BCL	**VARIABLE OUTPUT IN BCL**
&EBCDIC	**VARIABLE OUTPUT IN EBCDIC**
&DECIMAL	**VARIABLE OUTPUT IN DECIMAL**
&HEX	**VARIABLE OUTPUT IN HEX**
&OCTAL	**VARIABLE OUTPUT IN OCTAL**
&TYPE	**VARIABLE OUTPUT ACCORDING TO ITS DECLARATION IN THE USER'S PROGRAM.**

ADDRESS COUPLE OUTPUT

&ADDRESS
Outputs the address couple of the last identifier.

VALUE MODIFICATION

&ALTER
Changes the value of the last real, integer or Boolean identifier looked at in the identifier mode to <*val*>. <*val*> must presently be an integer. Arrays and pointers may not now be modified.

Example commands

```
&HEX  EBC  OCTAL  T
&ALTER  -465
```

An example **BREAKPOINT** conversation would be as follows. **BREAKPOINT** replies have an * on the left. Values refer to the example program above.

```
*   BREAK  @    4016000
    R
*   R  =
*   0.0
    /CONTINUE+3
*   BREAK  @    4019000
    /CONTINUE  4010000
*   BREAK  @    4010000
    R
*   R  =
*   19088743.0
    /HEX  EBCDIC
    R
*   R  =
*   EBC  ??????
*   HEX  000001234567
    A[0-1]
*   A[0-1] =
*   EBC  ABCKAB  ABCKAB
*   HEX  C1C2C3D2C1C2  C1C2C3D2C1C2
    /TYPE
    D[0,4-7]
*   D[0,4-7]  =
*   0.0  0.0  474.0  0.0
    &HEX  BCL  OCTAL
*   HEX  000000000000  000000000000  0000000001DA  000000000000
*   BCL  00000000  00000000  0000007.  00000000
*   OCT  0000000000000000  0000000000000000  0000000000000732
*   OCT  0000000000000000
    &ADDRESS
```

```
*   D IS (2 , 2)
    P8 FOR 11
*   P8 FOR 11
*   ABCKABABCKA
    &
*   EIGHT-BIT POINTER INDEX = 0 + 0
    BOO
*   BOO = TRUE
    EB [2] FOR 3
*   CDE
    S
*   S = 66.0
    &ALTER -79
*   S = -79.0
    /WHERE
*   BREAK @ 4010000
    /CONTINUE+1111
```

B7700TOG (RESET)

The **B7700TOG** option causes optimized code to be generated for the B7700 system. When running on B6700 system software, it is reset by default.

CHECK (RESET)

The **CHECK** option causes sequence errors to be flagged on both the **TAPE** and **NEWTAPE** files. If the sequence error occurs on the **TAPE** file, the message **SEQERR** followed by the sequence number of the last source image is printed at the right-hand side of the source image on the printout. If the sequence error occurs on the **NEWTAPE** file, the message on the printout is **NEWTAPE SEQ ERROR** followed by the sequence number of the last source image, and the message **NEWTAPE SEQ ERR** is displayed on the **SPO.** On a **CANDE** file, the sequence number of the card that caused the sequence error and the sequence number of the previous source image appear on the line following the source image.

Note that if **NEW** is not **SET** and resequencing is occurring, the old sequence number is the sequence number that is used.

CODE (RESET)

The code option causes the printout to contain the compiler-generated object code. **LIST** must be **SET** to produce the printout.

DOUBLESPACE

The **DOUBLESPACE** option must be **SET** during compilation of the compiler in order to get double space default; otherwise, single spacing is the default.

DUMPINFO (RESET)

Refer to **LOADINFO** option.

ENDSEGMENT (RESET)

Refer to **BEGINSEGMENT** option.

ERRLIST (RESET)

The **ERRLIST** option causes syntax error information for **CANDE** to be written on the **ERRORFILE** file. When a compilation error is detected in the source input, the offending line of text, an error message, and a pointer to the syntactical item in question are written on two lines in the **ERRORFILE** file. This option is provided primarily for use when the compiler is called from a remote terminal by the **CANDE** language, but it can be used regardless of the manner in which the compiler is called. When the compiler is called from **CANDE**, the default state of the **ERRLIST** option is **SET** and **ERRORFILE** is automatically equated to the remote device involved.

EXTERNAL (autobinding only)

The **EXTERNAL** option causes designated *<program unit>*s to remain external to the program. (**BINDER** will normally attempt to bind all external *<program unit>*s.)

FORMAT (RESET)

If the **FORMAT** option is **SET** while the **LIST** option is **SET**, several blank lines are inserted after each procedure in the input printout to aid readability.

GO TO (cannot be **SET** or **RESET**)

This compiler option, when used, should appear with no other options on a option control card and must not be preceded by an *<option action>*. This option is intended for use with symbolic disk files. It does not work on tape files.

Syntax

<go specification> ::= *<go part>* *<sequence number>*

<go part> ::= **GO** |
GO TO

Semantics

The *<sequence number>* is the sequence number appearing on a card image in the **TAPE** file. The **GO TO** compiler option causes **TAPE**, the secondary symbolic input file, to be repositioned so that the next card image used from this file by the compiler is the first card image with a sequence number greater than or equal to *<sequence number>*.

This option cannot be used in a *<define declaration>* or in **INCLUDE**d text.

The **TAPE** file must be properly sequenced in ascending order; that is, each sequence number on each card image in the file must be greater than the preceding sequence number. One can "go to" a lower sequence number. This sequencing method is necessary because a "binary search" is performed to find the *<sequence number>*.

HOST

The **HOST** option specifies the title of the host file. This option is always **SET**, except for intrinsic binding. Refer to the **AUTOBIND** option. (Refer to the B 7000/B 6000 Program Binder Reference Manual.)

INCLNEW (RESET)

If both the **NEW** and **INCLNEW** options are **SET**, **INCLUDE**d text is used as output to the **NEWTAPE** file. If the **INCLNEW** option is **RESET**, the **INCLUDE**d text is not used as output to the **NEWTAPE** file. However, **$INCLUDE** cards that are contained in the **CARD** and **TAPE** files are used as output to the **NEWTAPE** file when the $ sign is in column 2. If the **NEW** option is **RESET**, the state of the **INCLNEW** option is ignored.

INCLSEQ (RESET)

When **SET**, **INCLSEQ** resequences an included file (**SEQ** must also be **SET**). The **INCLNEW** option must be **SET** for an included file to have its card images in the **NEWTAPE** file.

INCLUDE (cannot be SET or RESET)

The *<include card>* is a special compiler control option that permits indirect source language input to the compiler from files other than the **CARD** and **TAPE** files. The user can specify on these cards that portions of other files are to be included in the source language input. The **INCLUDE**d card images are compiled in place of the *<include card>*. It is possible that the **INCLUDE**d card images can themselves contain **INCLUDE** cards, and in this way **INCLUDE**d text can be nested up to five levels deep. The blocking structure of the **INCLUDE**d files must follow the same rules required of the **CARD** file.

Syntax

<include card> ::= $INCLUDE *<file option>* *<start option>* *<stop option>*
<file option> ::= *<empty>* |
<internal name> |
<title> |
<label equating option>
<start option> ::= *<empty>* |
* |
<sequence number>
<stop option> ::= *<empty>* |
– *<sequence number>*
<internal name> ::= *<identifier>*
<title> ::= {quoted string containing file title}
<label equating option> ::= *<internal name>* = *<title>*
<sequence number> ::= {unsigned integer up to 8 digits long anywhere in columns 2 through 72}

Examples

A compiler control card that has an **INCLUDE** compiler control statement can contain no other control information other than that statement. Valid examples of **INCLUDE** compiler control cards are as follows:

$INCLUDE FILE8 00001000 – 09000000
$INCLUDE
$INCLUDE "SOURCE/XYZ" – 900
$INCLUDE *
$INCLUDE INCLFILE
$INCLUDE INCL = "SYMBOL/ALGOL/INCLUDE1."

Semantics

The *<file option>* specifies the file to be included. The *<start option>* specifies the sequence number of the first card image to be included from the file. The *<sequence number>* forms of the *<start option>* should be used only on properly sequenced files. The *<stop option>* specifies the sequence number of the last card image to be included from the file.

The first **INCLUDE** card example above instructs the compiler to accept as input at this point of the input file all records from the library file indicated by the *<internal name>* **FILE8** beginning with the card image with the sequence number 00001000, or the next higher sequence number, and terminating with the card image with the sequence number 9000000, or the preceding lower sequence number if no card image in the file has this number.

If the *<internal name>* is used, as in the first example, the name is used for purposes of label-equating. The *<title>* of the **FILE8** file can be specified on a **FILE** system control card. This card follows the **COMPILE** control card, which initiates the compilation. For example,

<I>FILE FILE8 (TITLE = SOURCE/INPUT/INCL)

The preceding card indicates that the *<title>* of the included file is **SOURCE/INPUT/INCL**; the file is a disk file, since the **FILE** card in this example does not change the value of the **KIND** attribute from the default specification.

If the *<title>* option is used, as in the third example, the string specifies the title of the actual file to be included. The *<internal name>* allows greater flexibility than *<title>* because the actual file name of the included file can be altered by simply changing the label-equating (file) card at the beginning of the program deck.

If *<file option>* is *<empty>*, as in the second example, then the same file as the one specified on the previous *<include card>* at the same level of nesting is included. Therefore, the first *<include card>* at any of the five possible levels of nesting must contain either an *<internal name>* or a *<title>*.

The second example instructs the compiler to accept as input at this point of the input file a portion of the file accessed by the last **INCLUDE** card in this deck at this level. This card images to be included consist of all records remaining in the included file following the last record of that file accessed by the preceding **INCLUDE** card referencing that file. If, for example, the example card is in the same deck as the first example card and no other **INCLUDE** cards intervene, the *<include file>* that is accessed is the **FILE8** file, and the first record that is included is the card image with the next sequence number after 09000000.

The third example instructs the compiler to accept as input at this point of the input file a portion of the file with the *<title>* of **SOURCE/XYZ**. The card images to be included consist of all records remaining in the file between the last record included from that file at this level and the record with the sequence number immediately higher than 00000900.

If the asterisk (*) form of the *<start option>* is used, inclusion begins at the point at which it left off the previous time that inclusion took place on the file at the particular level of nesting. For example, if the fourth example card is in the same deck as the third example card and no other **INCLUDE** cards intervene at the particular nesting level, the records that are included are the remaining card images in the **SOURCE/XYZ** file up to the card with the sequence number immediately higher than 00000900.

If the *<start option>* is *<empty>*, the inclusion begins with the first record of the file. And, if the *<stop option>* is *<empty>*, then the last record of the file is the last record included. When the *<start option>* and/or the *<stop option>* are used, the sequence numbers of the file must be in ascending order.

The fifth example instructs the compiler to accept as input at this point of the input file a portion of the file with the internal name **INCLFILE**. The card images that are included consist of all records remaining in the file between the last record included from that file at this level and the end of the file.

The final example illustrates how both *<internal name>* and *<title>* are specified on a **$INCLUDE** card. In this way a default title, "**SYMBOL/ALGOL/INCLUDE1.**", is associated with an *<internal name>*, **INCL**. This allows label equating various files. For example, the WFL statement <I> **ALGOL FILE INCL = SYMBOL/ALGOL/INCLUDE2** causes the file with the *<title>* "**SYMBOL/ALGOL/INCLUDE1.**" to be replaced by the file with the *<title>* **SYMBOL/ALGOL/INCLUDE2.**

Source files suitable for use by **INCLUDE** compiler control statements can be produced by the compiler via the **NEWTAPE** file.

INITIALIZE (autobinding only)

The **INITIALIZE** option is used in intrinsic binding for the purpose of allowing non-**ESPOL** intrinsic to refer to **MCP** procedures with fixed addresses.

INSTALLATION (RESET)

If the **INSTALLATION** option is **SET**, installation intrinsics can be referenced by the program. This option must be **SET** before the beginning of the program. For example, before the first **BEGIN** in a block program, before global declarations in a separately compiled procedure with global declarations, or before the first procedure in separately compiled procedures without globals, setting this option at any other time has no effect.

SETting the **INSTALLATION** option causes the compiler to search the **MCP** intrinsic directory for all intrinsics that can be referenced by an **ALGOL** program. It puts the identifiers for all such intrinsics into the **INFO** file table of the compiler.

Syntax

<installation compiler option> ::= **INSTALLATION** *<number-list>*

<number-list> ::= *<number-element>* |
<number-element> *<number-list>* |
<number-element> , *<number-list>* |
<empty>

<number-element> ::= *<install-no1>* |
<install-no1> – *<install-no2>*

Semantics

<install-no1> and *<install-no2>* are unsigned integers between one and 2047, inclusive. *<number-element>s* must be in ascending sequence, with no number repeated.

Note that an **INSTALLATION** compiler card with no *<number-list>* is equivalent to the *<number-list>* 100-2047.

The installation intrinsics that are loaded are either those included in a range or explicitly stated on the last installation setting encountered.

A syntax error is emitted if the *<number-list>* is not in ascending sequence, if any of the ranges specified overlap, or if the second number in a range is not larger than the first number. Numbers larger than 2047 are treated as if they are 2047.

INTRINSICS (RESET)

The **INTRINSICS** option causes compilation of a procedure at lex-level two and allows for global declarations, declarations enclosed in brackets preceding the procedure. Such global declarations are not normally allowed when compiling separate procedures to be used as installation intrinsics. The code-file title is the same as if compiled at lex-level three. Thus, the separate procedures being compiled can be bound afterwards into the intrinsic file. When used with the **BINDER** program for binding procedures to intrinsic files, **INTRINSICS** must be **SET** before the first source statement.

LEVEL (cannot be SET or RESET)

The *<outer level>* parameter controls the lexicographic (lex) level at which the compilation is to occur.

The proper format for this option is as follows:

LEVEL *<unsigned integer>*

where the *<unsigned integer>* corresponds to the desired lex-level number. The **LEVEL** option should not be preceded by an *<option action>*. This option allows the user to override the lexicographic levels assigned by the compiler. The default level is 2 for programs and 3 for separately compiled procedures. Only **LEVEL** option control cards that appear before the start of source language input are considered by the compiler.

LIBRARY (RESET)

When compiling multiple procedures, such as the intrinsics, it is more efficient to set the **LIBRARY** option. This causes all object program code, if more than one *<program unit>* is being compiled, to be put in one file and marked as a multiprocedure code file. Binder control cards for binding these procedures, either to a host or an intrinsic file, have to be changed, however. If, for example, some procedures were compiled as "A/B", then the **BIND** option card would have to be changed from:

BIND = FROM A/=;

to

BIND = FROM A/B;

The **LIBRARY** option is initialized to **TRUE** when compiling from **CANDE** or when using the **SEPCOMP** facility. (Refer to the B 7000/B 6000 Program Binder Reference Manual, form 5001456.)

LIMIT (cannot be SET or RESET)

The *<error limit>* parameter allows the user to control compiler error terminations. The proper format for the **LIMIT** option is as follows:

LIMIT *<unsigned integer>*

Compilation is terminated if the number of errors detected by the compiler equals the *<unsigned integer>*. A limit of zero (0) indicates than an "infinite" number of errors are to be allowed. The **LIMIT** option must appear only on an option control card that precedes the first source language input statement. If no **LIMIT** statement appears, a default error limit of 150 is assigned unless the compilation is initiated through **CANDE**, in which case the default error is 10.

LINEINFO (RESET)

The **LINEINFO** compiler option provides source line identification information at the point in a program where an error has occurred. The option saves sequence or line number information at compile time and its relation to the code emitted at compile time. Because an additional significant amount of disk storage may be required in the code file of a compiled program it is not desirable to **SET** the option for "debugged" programs.

LIST (SET; RESET for CANDE and BINDER)

The **LIST** option causes a printout to be generated on the compiler output **LINE** file. The contents of such printouts are specified in the preceding paragraphs describing compiler features. If the **LIST** option is **RESET**, only syntax error messages are listed.

When the **LIST** option is never **SET** for a compilation, that is, for non-**CANDE** compilations; or when the **LIST** option is **RESET** by an option control card, that is, the first card of the input deck, a printout can be genereated by **SET**ting the **TIME** compiler option. The printout contains only compilation trailer information.

LISTDELETED (RESET)

When **SET**, the **LISTDELETED** compiler option causes the inclusion in the printout of card images from the secondary input file **TAPE** that are replaced, voided, or deleted during the compilation. Four asterisks appear on the printout to the left of each of these source images. The following words appear to the right of the source images: **REPLACED**, if the source image is replaced by a card in the primary input file; **VOIDT**, if the card image is voided from the input file by the **VOIDT** compiler option; or **DELETED**, if the card image is deleted by an option control card with a $ sign in column 1 and no option action, options, or parameters.

LISTINCL (RESET)

The **LISTINCL** compiler option controls the printout of cards from included files. The **LIST** and **LIST-INCL** compiler options must both be **SET** if a printout of the included file is desired.

A page eject is suppressed in an included file if the **PAGE** option is present and if the **LIST** and **LISTINCL** are not both **SET**.

LISTOMITTED (SET)

When the **LISTOMITTED** option is **RESET**, source code cards being **OMIT**ted will not appear on the printout. However, the **SET**ting and **POP**ping of dollar cards will be printed if the **LIST** option is **SET**.

This option is designed to aid in following program logic where many combinations of **OMIT**ting are frequently used.

LISTP (RESET)

When **SET**, the **LISTP** option causes patches and input records from the compiler **CARD** file to be included on the printout while records from the compiler **TAPE** file are excluded. This option is effective only if the **LIST** option is **RESET**. If the **LIST** option is **SET**, the state of **LISTP** is ignored. Therefore, the **LISTP** or the **LIST** option causes a printout to be generated when **SET**.

LOADINFO (RESET)

The **LOADINFO** and **DUMPINFO** options enable the user to save or load the contents of the main table in the compiler via the file. The tables saved included **INFO**, **ADDL**, **TEXT**, and **STACKHEAD** arrays, plus several simple variables.

The options are used in conjunction with the separate compilation of procedures. Typically, all global declarations are compiled and then the tables are dumped to file **INFO**. Subsequent compilations of procedures merely load the **INFO** file and go to the start of the procedure symbolic. For example,

```
<I>COMPILE MAKE/INFOFILE ALGOL LIBRARY
<I>ALGOL FILE INFO=MY/INFO
<I>DATA
    [
    {global declarations}
    $ DUMPINFO
    ]
    END.
<I>END
<I>COMPILE MY/PROGRAM ALGOL LIBRARY
<I>ALGOL FILE INFO=MY/INFO
<I>DATA
    %%% LOAD THE GLOBALS INTO TABLES.
    [
    $ LOADINFO
    {additional global declarations}
    ]
    {separate procedure declaration}
<I>END
```

In order to facilitate their use with intermediate level global binding, the **DUMPINO** and **LOADINFO** options can be followed by either an internal file name or an external file name and terminated with a period enclosed in quotes. This file name information is in a format similar to the **INCLUDE** option. This permits selective **INFO** file dumping at several points and selective **INFO** file loading more than once throughout a compilation.

The **DUMPINFO** and **LOADINFO** compiler options must be the last options appearing on an option card.

When a new **LOADINFO** operation is done, all old **INFO** file structures in **ALGOL** are removed. Thus, compiling different portions of the same program, even if they operate in different environments, can now be done in the same compilation.

The **LOADINFO** option changes all variables in the **INFO** file to globals and all procedures already compiled to be forward. This means that an **INFO** file created by a **DUMPINFO** operation that is done immediately before a procedure in a normal compilation is suitable for future use as globals if that procedure is to be compiled separately.

Caution is generally required only when variables with the same name are declared at different levels; a separate compilation can only access the last such variable seen before the **LOADINFO** operation occurred.

MAKEHOST (RESET)

An automatic separate compilation and binding facility is particularly helpful for development work on large **ALGOL** programs, since the amount of control information required by the compiler to replace procedures in host programs has been reduced to a minimum (refer to **SEPCOMP** option). This facility is intended as a supplement to, not a replacement for, other methods of compiling and binding **ALGOL** procedures. Given only the name of the host program to be changed, and the patches to change it, the compiler is able to separately compile and bind to the host program only those procedures which are being changed. This method requires that certain information be associated with the host program, information that is not normally collected and saved during the compilation of a program.

MAKEHOST requests that this information be saved when compiling a block program or procedure at level two. This option cannot be explicitly referenced after the appearance of the first syntactical item.

If **MAKEHOST** is **SET**, information is saved in the code file of the program about the symbolic file used or created by the compilation, the sequence ranges of all procedure bodies declared in the outer block of the program, and the declarative environment of the outer block. The environment of the outer block is similar to the information obtained by the **DUMPINFO** dollar option, and enables level three procedures to be compiled separately within this environment.

Additional environments may be saved, if desired, in order that procedures at levels greater than three may be replaced. Additional environments can only be specified immediately following the setting of the **MAKEHOST** option.

Syntax

<additional-environment> ::= *<empty>* | (*<environment-list>*)

<environment-list> ::= *<environment>* | *<environment-list>*, *<environment>*

<environment> ::= *<procedure-identifier>* | *<procedure-identifier>* OF *<environment>*

Examples

```
$ SET MAKEHOST
$ SET MAKEHOST (PASSONE, PASSTWO)
$ SET MAKEHOST (PASSONE, PASSTWO, WRAPUP OF PASSTWO)
```

In the example, the last dollar card overrides the first two, saving the environments of **PASSONE, PASSTWO** and **WRAPUP OF PASSTWO** in addition to the environment of the outer block. The *<environment-list>* may extend across several card images, a precedent arising more out of necessity than desire. The current implementation, for reasons of simplicity, requires environments to be fully qualified through level three procedure identifiers. If an environment is never found during the course of compilation, the compiler lists the unknown environment, along with a syntax error. Environments may appear in any order, without regard to the actual block structure of the host program.

Finally, when making a host program, **SET** the **NEW** dollar option if there are any changes to the host program. The default symbolic file associated with the host program is the title of the **NEWTAPE** file if one has been created, otherwise it is the title of the **TAPE** file.

MCP (RESET)

The **MCP** option causes all value arrays, translate tables, truthsets, and constant pools to be allocated at level 2. It must be **SET** before compiling the first syntactical item of a program.

MERGE (RESET)

When **SET**, the **MERGE** compiler option causes primary input, **CARD** file, to be merged with secondary input, **TAPE** file, to form the total input to the compiler. If matching sequence numbers occur, the primary input overrides. If the **MERGE** option is **RESET**, only primary input is used and secondary input is totally ignored. Therefore, the total input to the compiler when the **MERGE** option is **SET** consists of all card images from the **CARD** file, all card images from the **TAPE** file that do not have sequence numbers that can be found on cards in the **CARD** file, and all card images inserted into the text in these files by **INCLUDE** control cards. Card images in the **CARD** file also override **INCLUDE** compiler control cards in the **TAPE** file if conflicts in sequence numbers are encountered.

NEW (RESET)

When the state of the **NEW** option is **SET**, the merged input from the **CARD** and **TAPE** files is placed on the updated symbolic output file **NEWTAPE**. This file is coded in **EBCDIC** and is structured in 14-word records and 420-word blocks. Therefore, it can later be used as input to the compiler through the **TAPE** file. Text inserted into the **CARD** and/or **TAPE** files is placed in the **NEWTAPE** file if the **INCLNEW** compiler option is **SET**. Otherwise, if the **INCLNEW** option is **RESET** and the **NEW** option is **SET**, the **INCLUDE** cards are placed on the **NEWTAPE** file rather than the **INCLUDE**d text. All option control cards other than the **INCLUDE** cards in the merged **CARD** and **TAPE** file input are placed on the **NEWTAPE** file when **NEW** is **SET** and only if the initial $ sign on these cards is in card column 2.

The **NEW** option can be **SET** and **RESET** as necessary by option control cards appearing at any point in the input file. Such option control cards can also be placed on the **NEWTAPE** file if the $ signs on these cards are in column 2.

The contents of the **NEWTAPE** file can be monitored as follows: When both the **NEW** and the **LIST** options are **SET**, the **NEWTAPE** file contains all the source language statements that the **LINE** file contains, depending upon the state of the **INCLNEW** option, and all option control cards appearing in the **LINE** file that have their initial $ sign in card column 2. **INCLUDE** option control cards rather than **INCLUDE**d file text are placed in the **NEWTAPE** file when the **INCLNEW** option is **RESET**, but the **INCLUDE**d text is placed in the **LINE** file regardless of the state of the **INCLNEW** option.

The **NEWTAPE** file is created despite the occurrence of syntax errors in the source input. This file can be used as a secondary input for a later compilation or as an **INCLUDE**d file.

The **NEWTAPE** file can be label-equated so that, for example, the output goes to magnetic tape.

NEWSEQERR (RESET)

The **NEWSEQERR** option causes all non-ascending sequence record numbers on the **NEWTAPE** file to be flagged (equal record numbers are flagged). If sequence errors occur and the **NEWSEQERR** option is **SET**, the **NEWTAPE** file is not locked, the message **NEWTAPE NOT LOCKED** is displayed on the **SPO**, and the message **NEWTAPE NOT LOCKED** *<number of errors>* **NEWTAPE SEQUENCE ERRORS** is printed on the printout. The **NEWSEQERR** option is effective even if the **CHECK** option is **RESET**.

NOBINDINFO (RESET, autobinding only)

When **SET**, the **NOBINDINFO** option causes the binder to purge all binder information from the resultant code file. The resultant code file cannot then be used as input to the binder.

NOSTACKARRAYS (RESET)

When **SET**, the B 7700 **NOSTACKARRAYS** option suppresses the allocation of arrays with the stack, that is, it prevents short arrays from being allocated within the stack.

NOXREFLIST (RESET)

The **NOXREFLIST** compiler option and the **XREF** compiler option, when **SET**, prevent **SYSTEM/XREFANALYZER** from being initiated by the compiler. (The **NOXREFLIST** option has no effect if **XREF** is not **SET**.) The file **XREF/***<code file name>* is created where *<code file name>* is the name of the code file generated by the compiler. **SYSTEM/XREFANALYZER** can then be run using **XREF/***<code file name>* as described under the **XREF** compiler option.

The **NOXREFLIST** compiler option makes possible the label equating of **XREF** output to printer backup tape or the combining of **XREF** output with the rest of the job output.

The following example shows the label equating of **XREF** output to printer backup tape.

```
<I>   RUN  SYSTEM/XREFANALYZER  (132);
      FILE  XREFFILE  (TITLE = XREF/CODEFILENAME);
      FILE  LINE  (KIND = PRINTER, BACKUPKIND = TAPE)
```

OMIT (RESET)

The **OMIT** option causes card images from both the **CARD** and the **TAPE** files, other than $ cards, to be ignored, that is, they can be listed and/or written on a new symbolic file but not compiled. On the printout they are flagged by the word **OMIT**.

The **OMIT** $ card, when **SET**, causes $ cards in columns 3 through 72 that would otherwise be processed to be ignored. However, $ cards with the $ sign in columns 1 and 2 will continue to be processed. This permits flexibility in nested omits.

OPTIMIZE (RESET)

When **SET**, additional analysis of Boolean expressions used for conditional branches is performed, and code is generated to permit early termination of the expression evaluation. That is, **AND** and **OR** operations become conditional branches. For example, the code generated when **OPTIMIZE** is **SET** and **RESET** is as follows:

If A = B AND C = D THEN

SET		RESET	
VALC	on A	VALC	on A
VALC	on B	VALC	on B
EQUL		EQUL	
BRFL-LINK		VALC	on C
VALC	on C	VALC	on D
VALC	on D	EQUL	
EQUL		LAND	

PAGE (cannot be SET or RESET)

The **PAGE** compiler option must appear on an option card without an option action preceding it. When a **PAGE** option card appears, the printout is spaced to the top of the next page, but only if the **LIST** option is **SET**.

PURGE (autobinding only)

The **PURGE** option causes all input files specified in the *<file list>* to be removed from the disk directory after binding. Only files opened by the binder will be purged.

SEGDESCABOVE (RESET)

The **SEGDESCABOVE** option is used when compiling large programs which have difficulty addressing the segment dictionary.

Syntax

<segdescabove card> ::= $ *<option action>* *<number part>*

<number part> ::= *<empty>* | *<unsigned integer>*

When the compiler is compiling a host, this option causes all segment descriptors to be allocated starting at the specified D1 slot. Numbers must be in the range from 3 to 4096; if no *<unsigned integer>* is given, a default value of 2048 is assumed. This option is intended for generating host files and is ignored when compiling separate procedures (ones that are bound into a host file). The option is invalid for batch and cannot be modified once a compile has begun. The **BINDER** recognizes this special host and preserves the **SEGDESCABOVE** specification. Care should be taken when using this option as nonsegment descriptors may not fill the space below the segment descriptors; these unused slots occupy "save" memory when the program us running.

SEGS (RESET)

The **SEGS** option causes the printout to contain the beginning and ending segment messages. Note that setting the **LIST** option also sets the **SEGS** option. Therefore, **$ RESET SEGS** must follow **$ SET LIST**.

SEPCOMP

SEPCOMP invokes the automatic separate compilation and binding facility (refer to **MAKEHOST** option). As a dollar option, **SEPCOMP** has some peculiar distinctions. It cannot be explicitly referenced after the beginning of the compilation nor are multiple **SEPCOMP** option settings allowed since, when first **SET**, it initiates the preprocessing of the card file input. The title of the host program can be specified either as a string immediately following the word **SEPCOMP** on the dollar card or by label equating the **ALGOL** compiler's **FILE HOST**. The optional string specification has precedence over label equation. The following compile decks both specify a host file titled "A/HOST".

Examples

(deck 1)

```
?COMPILE A/B  WITH  ALGOL  FOR  LIBRARY
?DATA
$  SEPCOMP  "A/HOST"  LIST  STACK
$  SET  LINEINFO
   %  PATCH  CARD                    <seq-number>
?END
```

(deck 2)

```
?COMPILE  A/B  WITH  ALGOL  FOR  LIBRARY
?ALGOL  FILE  HOST=A/HOST
?DATA
$  SEPCOMP  LIST  STACK
$  SET  LISTDELETED
     %  PATCH  CARD                  <seq-number>
?END
```

Once the host file title is known, the patch cards must be provided. Dollar cards with blank sequence numbers are accepted following the dollar card setting the **SEPCOMP** option and prior to the first "patch card." A patch card is a card having a non-blank sequence number, at least one is required. Among patch cards having non-blank sequence numbers, sequence errors are not allowed. **SEPCOMP** examines the patch cards, decides which procedures can be compiled, and takes care of generating binder control information for replacing these procedures in the host programs. **SEPCOMP** always tries to compile procedures at the highest possible lex level. Therefore, the number of extra environments specified when making a host program has an effect on choices available to **SEPCOMP**.

SEPCOMP sets several other dollar options automatically in an effort to simplify operation. The **MERGE** option is unavailable for use during **SEPCOMP** control. References to this option are ignored after **SEPCOMP** has been **SET**. Setting the **MERGE** option prior to setting **SEPCOMP** is illegal since it destroys the default label equation of the symbolic file to be merged with the patches. The title of the default symbolic file is associated with the host program, but this title can be overridden by label equation of the **ALGOL** compile file TAPE. **SEPCOMP** SETS both the **AUTOBIND** and **LIBRARY** options, causing all procedures to be compiled into one multi-program code file, a temporary file left open for the use of the binder. Explicitly resetting **AUTOBIND** will prevent the binder from being called and result in the code file being locked on disk if compiled for library. Explicitly resetting **LIBRARY** will cause each procedure compiled to be put in a separate and permanent code file. Binding may still occur, but at a somewhat slower rate. If procedures are put in separate code files, the title of the code file is determined in the standard way, with the procedure name replacing the last identifier from the title on the compile card. Procedures compiled at lex-level four and higher have the name of their environment used in the code-file name also. For example, if two level-four procedures are compiled having the same name but different environments, such as:

```
? COMPILE  A/HOST  WITH  ALGOL  FOR  LIBRARY
? DATA
$  SEPCOMP "A/HOST"
$  RESET  LIBRARY
   %  PATCH CARD  TO  Q  OF  PASSONE          <seq-number>
   %  PATCH  CARD  TO  Q  OF  PASSTWO         <seq-number>
?END
```

Two code files would be produced (A/PASSONE/Q and A/PASSTWO/Q) in addition to the new host file "A/HOST" assuming **PASSONE** and **PASSTWO** were specified as extra environments when "A/HOST" was made.

The special information associated with the host program is always copied over by the binder to the code file of the new program so it also may be used as a host, as in the previous example. This information is not, however, "updated" either by the binder or the compiler during the **SEPCOMP** process. It is possible for this information to come to inaccurately reflect the actual structure and content of the host program with which it is associated.

Because of the order of the code file, it is much faster to bind to a bound host than to an unbound host. For this reason, it may be advantageous to **SET AUTOBIND** when compiling a host program just to get the binder to rearrange the code file.

SEQ (RESET)

The proper format of the **SEQ** option is as follows:

SEQ *<sequence base>* *<sequence increment>*

If the **SEQ** option is **SET**, the printout and the new secondary source language file, **NEWTAPE**, contain new sequence numbers as defined by the *<sequence base>* and *<sequence increment>*. If the *<sequence base>* and *<sequence increment>* are unspecified, a base of 1000 and increment of 1000 are assumed.

This option has effect only when the **LIST** and/or **NEW** options are also **SET**. The sequence numbers that appear on the card images in these files when the **SEQ** option is **RESET** are identical to the sequence numbers on the corresponding cards in the input file.

When the **SEQ** option is **SET**, sequencing begins with the default sequence number 00001000 and continues in default increments of 1000. These default sequencing parameters can be overridden as follows: The appearance of a one- to eight-digit unsigned integer on a option control card is assumed to be a control statement associated with the **SEQ** option when this integer is not immediately preceded on the card by the option name **INCLUDE, LEVEL,** or **LIMIT.** This integer is employed as a sequencing argument when the state of the **SEQ** option is **SET.** If the integer is preceded by a plus sign (+), the integer is used as the sequence number increment size. Otherwise, the integer is used as the sequence number at which sequencing is to start as soon as **SEQ** is **SET**. Both of these arguments can be specified, overriding the default values of 1000.

The sequencing arguments can appear on the same option control card as that **SET**ting the **SEQ** option, on a preceding option control card, or on a later option control card. The following are examples of a sequencing argument appearing on the same option control card as the **SEQ** option:

$SET SEQ 100
$SEQ 20+1
$RESET CODE SET SEQ LIST +200

In the first example, sequencing of the **LINE** and **NEWTAPE** files begins at the sequence number 00000100 and continues in default increments of 1000 if no other sequencing increment is specified on a previous option control card. In the second example, sequencing begins at the sequence number 00000020 and proceeds in increments of 1. In the third example, if this is the first time **SEQ** is **SET** and no other initial sequence number has been specified on a previous option control card, sequencing begins at the sequence number 00001000, and proceeds in increments of 200. Otherwise, sequencing begins at a sequence number 200 greater than the last sequence number assigned, or at the initial sequence number assigned by an appropriate preceding option control card. An example of an option control card format that specifies sequencing arguments but not the **SET**ting of the **SEQ** option is as follows:

```
$ 100 + 100                                                     00005000
```

This compiler control card specifies that, when the state of the **SEQ** option is **SET**, sequencing begins with the sequence number 00000100 and proceeds in increments of 100. The standard option states are not affected because this card contains only parameter control information and no standard option names. If the **SEQ** option is **SET** when this control card appears, these two sequencing arguments take effect immediately.

SEQERR (RESET)

The **SEQERR** option causes sequence errors on the **TAPE** file to be flagged. If sequence errors occur and the **SEQERR** option is **SET**, the code file is not locked, the message **CODE FILE NOT LOCKED** is displayed on the **SPO**, and the message **CODE FILE NOT LOCKED** *<number of errors>* **TAPE SEQUENCE ERRORS** is printed on the printout.

<sequence base> and *<sequence increment>*

The *<sequence base>* option contains the sequence number that is assigned, if the **SEQ** option is **SET**, to the next source language card image that is used as output. After each card image is used as output, the *<sequence base>* is increased by the *<sequence increment>*.

SINGLE (SET)

The **SINGLE** option causes the printout to be single-spaced. When the **SINGLE** option is **RESET**, the printout is double-spaced. The default value of **SINGLE** is a compiler compile-time option.

STACK (RESET)

The **STACK** option causes the printout to contain relative stack addresses in the form of address couples for all program variables. **LIST** must be **SET**.

STATISTICS (RESET)

When **SET**, the **STATISTICS** compiler option gathers timing statistics. The option is examined at the beginning of each procedure or block, and if it is **SET** at that time, timing statistics are gathered for that procedure or block. Although the option may be altered at any time, only its status at the beginning of procedures and blocks is significant for determining whether timings are made.

If statistics are taken for a procedure or block, then the frequency of that procedure or block is measured, along with the length of time spent in that procedure or block. When the program is completed for any reason (including both normal **EOJ** and **DS**-ing), the statistics are printed out on the diagnostic file.

On the output listing, an asterisk (*) indicates that there is some doubt about the timings for the specific procedure name that precedes the asterisk. In addition, timings are invalid for any procedure or block that is entered by a "bad go to." Only the first six characters of any identifier are printed on the printout.

For any procedure or block that has statistics gathered, it is possible to break down the timings to the label level within that procedure by setting the option **LABELS. LABELS** must appear in parentheses immediately after the word **STATISTICS** on the option card. It may be preceded by **SET** or **RESET**. If both are omitted, **SET** is assumed. For example,

$ SET STATISTICS (LABELS)

will begin timing label breakpoints, and

$ SET STATISTICS (RESET LABELS)

will end timing of label breakpoints. **SET** or **RESET** inside the parentheses only has effect for the duration of the parentheses.

STOP (autobinding only)

The **STOP** option causes the **BINDER** to stop interpreting input statements and option cards, causing them to be flushed out.

TIME (RESET)

The **TIME** option causes trailer information, such as number of errors, number of segments, and compilation time, to be printed on the printout. The **TIME** option is **SET** by default when the **LIST** option is **SET.**

No source cards are listed, assuming the **LIST** option has been **RESET** for the entire input deck and no errors occurred. Therefore, since this trailer information is printed when the **LIST** option is **SET**, the state of the **TIME** option is significant only when the **LIST** option is **RESET.**

<user option> **(RESET)**

If a word on an option control card is not recognized as one of the *<option>s,* it is considered to be a *<user option>*. It can be manipulated exactly like any other option, that is, it can be **SET, RESET,** or **POP**ped.

Any option, standard or user, can be **SET** from any *<user option>* by using an equal sign. The format is **SET** *<option>* = *<option expression>*, where *<option expression>* ::= *<option>* | **NOT** *<option>*. For example,

```
$ SET MODULE
   .
   .
   .
$ SET OMIT = MODULE
$ SET LIST = NOT MODULE
```

The preceding example defines and **SET**s a *<user option>* called **MODULE.** Later, **OMIT** and **LIST** are **SET** and **RESET,** respectively, if **MODULE** is still **SET.** If a *<user option>* is not explicitly **SET**, it is **RESET** by default.

Boolean operations can be performed by setting the *<option>*s equal to *<Boolean expression>*s composed of * (itself), **EQV, IMP, OR, AND, NOT, TRUE, FALSE,** and *<user option>*s. For example,

```
$ SET OMIT = OPT1 AND OPT2 OR NOT OPT3
$ SET OMIT = * OR OPT4
```

The following illustration, figure D–2, shows the organization of a card deck that describes the method by which specific portions of source input code can be compiled and placed on a printout simply by setting a single user option.

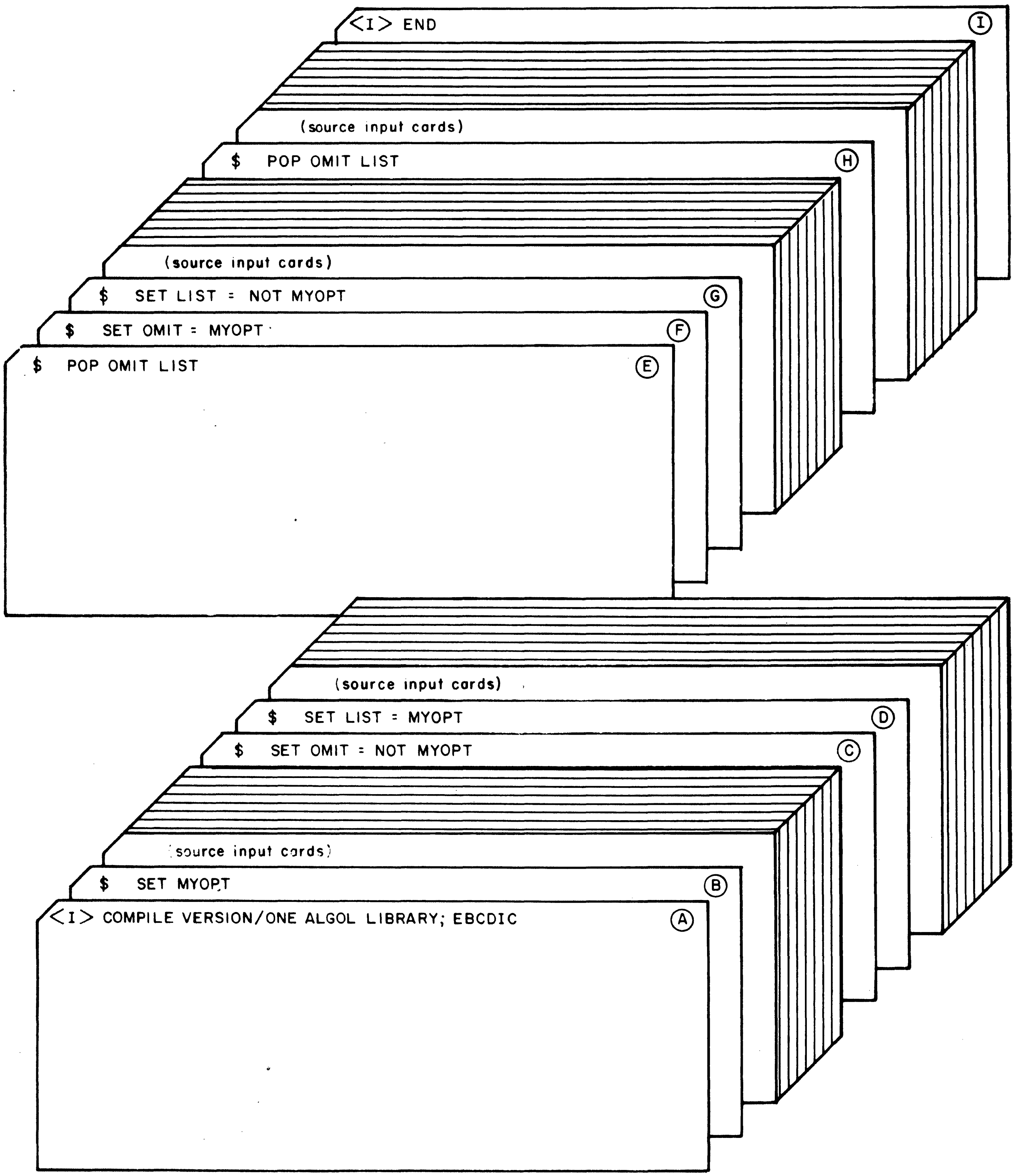


Figure D–2. Use of the Explicit **SET**

CARD	DESCRIPTION
(A)	The first card is an **MCP** system control card that contains the **COMPILE, LIBRARY** and **EBCDIC** control statements.
(B)	Card (B) defines and **SET**s a user option called **MYOPT**.
(C)	Card (C) **RESET**s or **SET**s the standard compiler option **OMIT** if **MYOPT** is **SET** or **RESET**, respectively.
(D)	Card (D) **SET**s or **RESET**s the standard compiler option **LIST** if **MYOPT** is **SET** or **RESET**, respectively.
(E)	Card (E) returns **OMIT** and **LIST** to their previous states.
(F)	Card (F) **SET**s or **RESET**s **OMIT** if **MYOPT** is **SET** or **RESET**, respectively.
(G)	Card (G) **RESET**s or **SET**s **LIST** if **MYOPT** is **SET** or **RESET**, respectively.
(H)	Card (H) returns **OMIT** and **LIST** to their previous states.
(I)	The final card is the **END** system control card.

In figure D–2, when **MYOPT** is **SET**, the source language information on the cards following (D) is included in the compilation and in the output printout. The source language information on the cards following card (G) is not included in either the compilation or printout.

Conversely, if **MYOPT** is **RESET**, accomplished by removing card (B), the information on the cards following card (D) is not included, and the information on the cards following card (G) is included in the compilation and printout.

The source language information on the cards immediately following cards (B) and (H) is included in both compilation and output printouts whether or not **MYOPT** is **SET**.

USE (autobinding only)

USE provides **BINDER** a technique for matching identifiers in a host with differing identifiers in a separate *<program unit>*.

VERSION (SET, RESET, and POP are ignored by the compiler)

The **VERSION** compiler option allows the user to specify an initial version number for a source program, to replace an existing version number, or to append an existing version number.

Syntax

<version compiler option> ::= *<replace version>* |
<append version>

<replace version> ::= $ **VERSION** *<version increment>* . *<cycle increment>*
<patch number>

<version increment> ::= *<2 digit integer>*

<cycle increment> ::= *<3 digit integer>*

<patch number> ::= *<empty>* |
. *<3 digit integer>*

<append version> ::= $ **VERSION +** *<version increment>*
. + *<cycle increment>* *<patch number>*

Examples

$ VERSION 25.010.010
$ VERSION +01.+001.010

When compiling with the **NEW** compiler option **SET** and a **VERSION** compiler card appears in the symbolic, and if the patch deck contains a *<replace version>* or *<append version>*, the new symbolic is updated to the version, cycle, and patch number on the last **VERSION** compiler card in the patch deck. The sequence number must be less than the one in the symbolic.

COMPILETIME(20), **COMPILETIME(21)**, and **COMPILETIME(22)** give the user the ability to access the version, cycle, and patch numbers, respectively.

VOID (RESET)

If the **VOID** option is **SET**, all input, other than $ cards, from the **TAPE** and the **CARD** files is ignored by the compiler until the **VOID** option is **RESET** or **POP**ped into a **RESET** state. The ignored input is neither listed nor included in the updated symbolic file regardless of the states of the **LIST** and **NEW** options. The **VOID** option can be **RESET**, once it is **SET**, only by a option control card in the **CARD** file.

VOIDT (RESET)

If the **VOIDT** option is **SET**, all secondary input, other than $ cards, from the **TAPE** file is ignored by the compiler until the **VOIDT** option is **RESET** or **POP**ped into a **RESET** state. Therefore, while the **VOIDT** option is **SET**, only primary input is compiled. The ignored input is neither listed nor included in the updated symbolic file regardless of the states of the **LIST** and **NEW** options. The **VOIDT** option can be **RESET**, once it is **SET**, only by an option control card in the **CARD** file.

WRITEAFTER (RESET)

The **WRITEAFTER** option implements the ability to write after carriage control. This option is **SET** around a *<file declaration>*, a *<switch file declaration>*, or an **I/O** statement. If **SET** around a *<file declaration>*, it pertains to all **I/O** statements where that file name explicitly appears. If **SET** around a *<switch file declaration>*, it pertains to those **I/O** statements explicitly using the switch **FILEID**. If **SET** around an **I/O** statement, it pertains to that particular **I/O** statement.

XDECS (RESET)

The **XDECS** compiler option, when **SET**, causes program declarations to be recorded for cross-referencing purposes when the **XREF** option is **SET**. This option is initially **SET** when the **XREF** option is **SET** and can be **SET**, **RESET**, or **POP**ped as many times as desired. The **XDECS** option is provided for use with the **XREF** option in order to select the portions of source code to be examined for information concerning declaration locations. **XREF** must be **SET**.

XREF (RESET)

When **SET**, the **XREF** compiler option causes, in the event of successful compilation, an index of all identifiers used in the compiled program to be written on the **LINE** file. This operation is accomplished by initiating the **SYSTEM/XREFANALYZER** program at the end of the compilation and giving to it a file containing the necessary information. These identifiers are arranged according to the **EBCDIC** collating sequence numbers of the card images on which the identifier appears. The **LIST** option need not be **SET** to generate a printout of this cross-reference information. The **XREF** option should be **SET** before any of the source input processed. This option cannot be **RESET** or **POP**ped once it is **SET**.

User options are also included when **XREF** is **SET**.

The line width, in characters, of the **XREF** output can be specified when the **XREF** option is **SET**. It is done as follows:

$ SET XREF *<optional unsigned integer>*

where *<optional unsigned integer>* can be in the range 72 ≤ *<optional unsigned integer>* ≤ 132.

If the optional unsigned integer is not specified, the default is taken as 132.

If the **SYSTEM/XREFANALYZER** Program is executed with control cards, the line width, in characters, must be specified as a parameter. For example,

<I>RUN SYSTEM/XREFANALYZER (*<line width>*);
FILE XREFFILE
(TITLE = XREF/*<codefilename>***)**;
END.

XREFFILES (RESET)

The **XREFFILES** compiler option, when **SET**, causes files to be saved for **SYSTEM/ INTERACTIVEXREF**. These files have the titles **XREFREFS/**<*code file name*> and **XREFDECS/**<*code file name*> where <*code file name*> is the name of the code file the compiler is generating.

When **XREFFILES** and **XREF** are both **SET**, the **SYSTEM/XREFANALYZER** run produces the files and the printed output. The **XREFFILES** compiler option has no effect if the **NOXREFLIST** compiler option is **SET** (i.e., files for **SYSTEM/INTERACTIVEXREF** are not generated).

Running **SYSTEM/XREFANALYZER** with a negative task value creates the same files as the **XREFFILES** compiler option.

XREFS (RESET)

The **XREFS** compiler option, when **SET**, causes program identifier references to be noted for cross-referencing purposes when the **XREF** option is **SET**. This option is initially **SET** when **XREF** option is **SET**, and can be **SET**, **RESET**, or **POP**ped as many times as desired. The **XREFS** option is provided for use with the **XREF** option to select the portions of source code to be examined for information concerning the location of identifier references. The only references that are cross-referenced are references of identifiers that are declared in portions of the source file where the **XDECS** option is **SET**. **XREF** must be **SET**.

$ (RESET)

When **SET**, the dollar sign ($) option causes the printout of all subsequent <*option control card*> images when the **LIST** option is **SET**. This option appears as **SET** or **$ $**.

APPENDIX E. PROGRAM SOURCE AND OBJECT FILES

COMPILER FILES

Compiler communication is handled through various input and output files (figure E–1). Cards, disk, or magnetic tape can be specified as source language input media. Input must be in the input format defined in the preceding sections. The compiler has the capability of merging, on the basis of sequence numbers, input from cards, tape, or disk. When inputs are being merged, indications of text insertions or replacements can be made to appear on the printout. In addition to the printout, the compiler can also generate updated symbolic files. These files can be created in addition to the compiler-generated output code file.

Input Files

The primary compiler input file is a card file with the internal name **CARD**; the secondary input file is a serial disk file with the internal name **TAPE**. The presence of the primary file **(CARD)** is required for each compilation; the presence of the secondary file **(TAPE)** is optional for each compilation. When two card images, one from the **CARD** file and the other the **TAPE** file have the same sequence number, the former is primary and is compiled, and the latter is ignored. This is the standard mode of handling source language input. File **CARD** can be either **BCL**-coded with 10-word records or **EBCDIC**-coded with 14-word records and can be either blocked or unblocked. File **TAPE** can be **BCL**-coded with 10-word records and 150-word blocks, or **EBCDIC**-coded with a 14- or 15-word record and 420- or 450-word blocks. Both the **CARD** file and the **TAPE** file can be label-equated (via the **FILE** system control card) to change the **TITLE** and **KIND** of the file. The **TAPE** file is used as input only when the **MERGE SEPCOMP** compiler option is **SET**.

Additional files can be used as input to the compiler through the use of the **INCLUDE** compiler option. These files may be either **BCL**-coded with 10-word records or **EBCDIC**-coded with 14- or 15-word records (similar to the **TAPE** file). These **INCLUDE**d files are, by default, disk files, and their internal names or titles are as specified on the **INCLUDE** compiler option cards.

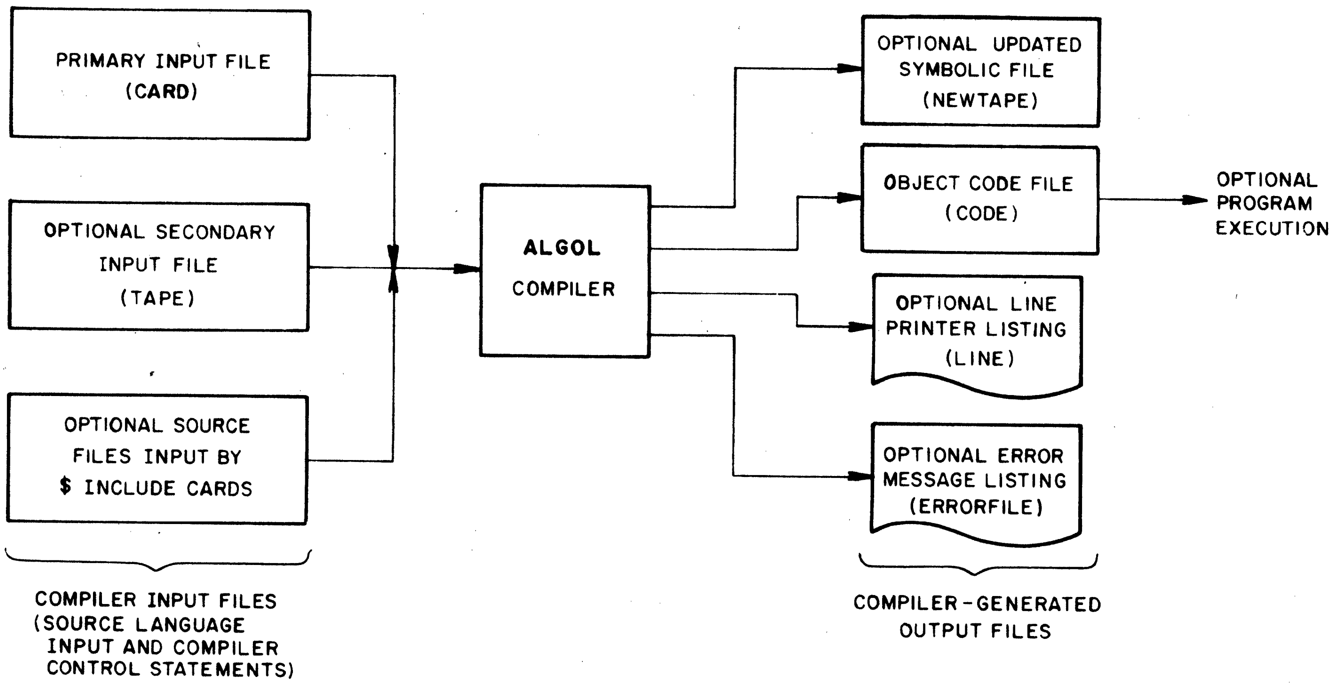


Figure E–1. ALGOL Compilation System

Output Files

Output files produced by the compiler consist of the object code file, an updated symbolic file, a line printer printout, and an error message file. The object code file has the internal name **CODE** and is saved on disk after the compilation unless the **COMPILE** system control card specifies compilation for syntax only, or unless syntax errors are detected in the source language input by the compiler. If compile-and-go is specified by the **COMPILE** system control card, then the object file is discarded after the code is executed. If compilation for library is specified, then the object code file is saved on disk. The title of the saved code file is identical to the program name appearing on the **COMPILE** system control card except in the case of separately compiled procedures. When subprograms are compiled separately, the title of the resultant object code library file consists of the program name appearing on the **COMPILE** system control card, with the right-most identifier in the program name replaced by the subprogram name. If there is only one identifier in the program name, this name is assigned as the code file title.

The compiled program is logically segmented within the resultant code file by program unit (procedure or subroutine). The code for each program unit begins at a physical disk segment boundary and fills as many disk segments as required within the limits of the system. An extremely long program unit may require more than one program segment, in which case it is automatically segmented as follows. (Such segmentation does not occur for separate compilations except for a main program.) As the amount of code generated for a program unit reaches 8192 words or 30*A words, where A is the numeric value of the **AREASIZE** of the code file, whichever is less, program unit segmentation occurs. The code for the program unit is then contained in two program segments. This segmentation can occur more than once, depending on the size of the program unit.

The updated symbolic file is, by default, a disk file generated only if the compiler option **NEW** is **SET**. This file contains the compilation source input or a selected portion of this input as specified by the states of the **NEW** and **INCLNEW** compiler options. It can be used as the **TAPE** file or an **INCLUDE**d file for a succeeding compilation. This output file has the internal file name **NEWTAPE** and contains **EBCDIC**-coded 14- or 15-word records in 420- or 450-word blocks.

The line printer printout is an optional print file that is created unless the compiler option **LIST** and **TIME** are **RESET**. (The **LIST** option is **SET** by default unless the compilation is initiated through **CANDE**.) The file has the internal name **LINE**, consists of 22-word **EBCDIC**-coded records, and contains the following information:

a. Source and compiler control statements used as input to the compiler.
b. Code segmentation information other than source input.
c. Error messages and error count.
d. Number of input card images scanned.
e. Elapsed compilation time.
f. Processing time required for the compilation.
g. Estimated size of the working stack when the program is executed.
h. Estimated size of the program files and related storage.
i. Total number of words of object code generated.
j. Stack address assignments.
k. Number of segments in the generated program.
l. Number of disk segments required for the program code file.
m. Estimated core storage required (in words).
n. Title of the generated code file.
o. Compilation date of the compiler.

Depending upon the specified setting of the **LIST**, **CODE**, **STACK**, and **TIME** compiler options, the line printer printout can contain more (or less) information than the basic items listed above. If either the **MERGE** or **SEPCOMP** compile-time option is set and the **LIST** or **LISTP** compile-time option is also set then card images from the **CARD** file are denoted on the printout by a P if they replace a card from the **TAPE** file and by a C otherwise. The C or P appears after the card contents.

The output error-message file with the internal file name and assigned title of **ERRORFILE** is an optional line printer file that is created when the **ERRLIST** compiler option is **SET**. This file is normally employed for compilations initiated through **CANDE**, in which case **ERRLIST** is **SET** by default and the **ERRORFILE** file is assigned to the remote device involved. The **ERRORFILE** file can also be used for compilations initiated through the card reader. This file is assigned **EBCDIC**-coded 12-word records that result in a a line width of 72 characters, allowing the file to be used as output to a remote terminal or card punch without truncation of text. When a syntax error is detected, the offending card image is written on this file with an error message and indicator of the syntactical item in question being written on the following line of text. The error message consists of an explanatory message and pointer (*) that indicates the probable location of the error. The asterisk is usually positioned one character past the "offender".

Compiler File Table

Table E–1, **ALGOL** Compiler Files, lists the internal name of the file, that is, the name used when the file is declared within the compiler, the purpose served by the file, the default **KIND** of the file, the code used to store file data, the default record size **(MAXRECSIZE)** and block size **(BLOCKSIZE)** of the file, and a brief commentary on the specific file. The attributes of any of these files can be changed by the use of **FILE** system control cards directed to the compiler.

Table E–1. **ALGOL** Compiler Files

INTERNAL NAME	PURPOSE	KIND	CODE	RECORD SIZE	BLOCK SIZE	COMMENTS
CARD	Input Card File	**CARD READER**	**EBCDIC** **BCL**	14 Words 10 Words	Blocked or Unblocked	Required for each compilation. Primary compiler input file; may be label-equated to change file attributes. **CANDE** file is equated to this file automatically for compilations initiated through **CANDE**. Default title is **CARD.** **BUFFERS = 5.** **FILETYPE = 7.**
TAPE	Input Disk File	**DISK**	**EBCDIC** **BCL**	14 or 15 Words 10 Words	420 or 450 Words 150 Words	Optional file; need not be present for for each compilation. Secondary compiler input file; selected as input by setting **MERGE** or **SEPCOMP** compiler option. Can be label-equated to change file attributes as desired. The default title is **TAPE**. **FILETYPE = 7.**
Can be specified on **INCLUDE** Compiler Control Card	Library Include File	**DISK**	**EBCDIC** **BCL**	14 Words 10 Words	420 Words 150 Words	Optional input file opened by $IN-**CLUDE** card that has appropriate file title or internal file name. Five levels (maximum) of nesting permitted. (Refer to discussion of **INCLUDE** option.) **FILETYPE = 7.**

Table E–1. **ALGOL** Compiler Files (Cont)

INTERNAL NAME	PURPOSE	KIND	CODE	RECORD SIZE	BLOCK SIZE	COMMENTS
CODE	Object Code File	**DISK**	Hexadecimal	30 Words	150 Words	Generated object code file. Saved or discarded and assigned a title as indicated by compilation method. For **CANDE** compilations, the title becomes: **USERCODE**/<*usercode*>/**OBJECT**/<*file-title*>. The default file title after compilation is the program name on **COMPILE** card (modified by subprogram **ID** when separate compilation is used).
NEWTAPE	Updated Symbolic Output File	**DISK**	**EBCDIC**	15 Words	450 Words	Optional output file produced when **NEW** compiler option is **SET**. This file contains portions of the source input and is label-equatable. It is suitable for use as a **TAPE** file for a later compilation. **BUFFERS** = 2. **AREASIZE** = 1000. **AREAS** = 60.

Table E–1. **ALGOL** Compiler Files (Cont)

INTERNAL NAME	PURPOSE	KIND	CODE	RECORD SIZE	BLOCK SIZE	COMMENTS
LINE	Line Printer Printout	**LINE PRINTER or DATACOM**	**EBCDIC**	22 Words	22 Words	Optional and label-equatable file. Produced when either the compiler option **LIST** or **TIME** is **SET**.
ERROR-FILE	Error Listing Output File	**LINE PRINTER**	**EBCDIC**	12 Words	12 Words	Optional error listing file produced when **ERRLIST** compiler is **SET**. Contains card images and error messages. Automatically provided for **CANDE** input.
		LINE PRINTER (KIND = 7)	**EBCDIC**	12 Words	12 Words	

APPENDIX F. BATCH FACILITY

INTRODUCTION

In certain situations, such as an educational environment, it may be possible to submit as a group a number of programs having the following characteristics:

a. The programs are to be compiled for syntax or compiled and executed.
b. If a program has no syntax errors and execution is requested, the execution time is relatively small.
c. Each program requires no more than one printer file and no more than one card reader file.
d. No program requires operator intervention.
e. No tasking is being used in the programs.

The batch facility is available so that groups of these programs may share the cost of many required system overhead functions normally associated with each job, such as job initialization and termination, linking with intrinsics, memory allocation, etc. Sharing these tasks promotes better use of system resources.

DECK SET-UP

Each job, which must conform to the restrictions discussed below, is set up as follows. All option control cards must have the format as described in Appendix D.

$JOB {*optional job title*}
    program
$entry
    data, if any

The first card in the deck is a **$JOB** card, which is used to indicate the beginning of the program. Optionally, a job title may appear on the **$JOB** card; this title is used to identify printed output produced by the compiler. Following the **$JOB** card is the source program, complete with the normal collection of option control cards, etc.

The next item is the **$ENTRY** card. This card is used to indicate the end of the program. It is also used to indicate that execution is desired if there are no errors. If the **$ENTRY** card is missing, the compiler assumes that execution of this program is not required, i.e., only a compile for syntax has been requested.

Finally, if the program uses a data block, then the data deck appears after the **$ENTRY** card. Note that the **$ENTRY** card is required for execution regardless of whether or not there is a data deck.

In addition, there are two compiler options which apply to the execution of each job. The two options, **PROCESSTIME** and **IOTIME**, perform the same functions as the corresponding system control cards.

For example:

$ PROCESSTIME = 2.2 PROCESSIOTIME = 5

would result in an upper limit of 2.2 seconds on execution time and 5 seconds in I/O time. There is no way to limit elapsed time, since programs cannot control the elapsed time.

If **PROCESSTIME** or **IOTIME** is set prior to the first **$JOB** card, the values become upper limits to the **PROCESSTIME** and **IOTIME** for individual user programs. Any individual user may be restricted to lower limits by the inclusion of **PROCESSTIME** or **IOTIME** option control cards following the **$JOB** card.

USING THE BATCH FACILITY

The job decks are placed together to form a single deck. This combined deck is preceded by the usual system control cards invoking the **ALGOL** compiler and followed by the **<I>END** system control card. This single deck is then processed by the system. The output produced is a single printer file having each compiler printout followed by its corresponding printed output.

Example of a batch of three **ALGOL** jobs:

```
<I> COMPILE BATCHX ALGOL; DATA
$ PROCESSTIME = 5.5 IOTIME = 1
$ JOB        ONE
$ SET LIST SEQ
BEGIN
FILE READER(KIND = READER);
INTEGER I;
ARRAY Z[0:2] ;
READ (READER,/, FOR I := 0 STEP 1 UNTIL 2
         DO Z [I] );
END.
$ ENTRY
2.315, 3.71828, .5772,
$ JOB       TWO
BEGIN
INTEGER A.B.C;
A := B + + C;  % A SYNTAX ERROR
END.
$ ENTRY
$ JOB    THREE
BEGIN
LABEL AGAIN;
REAL X;
AGAIN: X := 355/133;
GO TO AGAIN;
END.
<I> END
```

NOTE

No job in this three-job batch will be allowed more than 5.5 seconds of processor time (beware of job three!), nor more than 1 second of I/O time. Note also that job #1 will compile correctly and run, using the one-card data deck immediately following its **$ENTRY** card; that job #2 has a syntax error, but would have run had it been error free (it has no data deck, however); and that job #3, despite being syntactically correct, will not run without an **$ENTRY** card.

RESTRICTIONS

Each of these restrictions must be met in order for a job to be a candidate for successful processing by the batch facilities of the **ALGOL** compiler.

a. The printer file may not be explicitly opened, closed, or have its attributes changed. Attempts to do this will terminate the job.
b. Binding is not allowed. No compiler option pertinent to binding is valid.
c. Missing functions are fatal errors.
d. The <*wait statement*> is allowed but has no effect.
e. Any program action requiring operator intervention is a fatal execution error.
f. Only one printer file is allowed. If two or more printer files are declared, their output will be joined into a single file. Similarly, at most, one reader file is allowed.
g. The following compiler option cards are invalid or ignored when using the batch facilities: **ERRLIST, LIBRARY, LINEINFO, NEW, TIME, XREF.**
h. All job decks must be punched with the same card codes **(EBCDIC, BCL,** or **ASCII).** The system control card preceding the job decks **(DATA, EBCDIC, ASCII,** or **BCL)** must also indicate the proper card code.
i. Tasking is disallowed.

IMPLEMENTATION SCHEME

The goal of the implementation has been to eliminate as much normal system overhead as possible by reducing the number of tasks initiated in the system within the natural running environment of the B 7000/ B 6000 Information Processing System.

In order to eliminate many initiations of a compiler, the individual jobs are collected into a batch and presented as one file to the Batching Compiler. This obviously reduces the number of compiles to one compile, enabling the compiler to "get up to speed". The compilation process for each individual job is virtually the same as for non-batched jobs and yields equally efficient object code. When the compiler finishes compiling the last individual job, it generates special object code in the outer block to link each individual job to the next one. Should any individual job have syntax errors, or specify **COMPILE FOR SYNTAX,** it is not linked into the other jobs. The code of all individual jobs resides in one code file or disk at the end of the compile.

The printed output from the compiler is directed to a backup disk file with an altered **BDBASE** so that it will not be printed by **AUTOPRINT.** Logging information regarding the compile is also saved in this file. The execution of the code is:

1. Build the D2 stack.

2. Call **BATCHMONITOR** passing it a procedure which serially calls each individual job.

3. **BATCHMONITOR** processes the procedure passed to it. If any job should cause a fatal execution error, **BATCHMONITOR** reprocesses the procedure, which sequences automatically to the next individual job.

4. **BATCHMONITOR** rewinds the two backup printer files, extracts the logging information, and collates the output into a new printer file.

5. Return to the MCP.

The compiler makes attempts to share arrays from one individual job with succeeding jobs, eliminating many presence bit interrupts. Additionally, all jobs share the same printer file and intrinsics and may even share the same code segments. The individual jobs run serially and share the same stack space.

One job is protected from previous jobs by the **BATCHMONITOR**, a **DCALGOL** intrinsic. Should one job have an error, the execution is reinitiated by **BATCHMONITOR** at the next individual job. Should a job use an excessive amount of either I/O time or processor time, this fact is noticed by **BATCHMONITOR**, and the individual job is terminated. Likewise, **BATCHMONITOR** enforces the rule that no RSVP messages are allowed, by terminating the job that causes one.

BATCHMONITOR extracts the logging information from the two printer files and summarizes it at the end of the output of each individual job. This is easily modified to interface with the accounting system of a given installation. Additionally, two words in each log record furnished by the compiler and the individual job are spares to facilitate any installation extensions.

APPENDIX G. RUN-TIME FORMAT ERROR MESSAGES

The meanings of the various format error numbers pertaining to free-field input are as follows:

NUMBER	ERROR MESSAGE
400	An error on input occurred when the intrinsic did a logical I/O.
416	Attempted recursive I/O – evaluation of a list element caused a READ/WRITE/CLOSE on the current file.
420	Input from the *<core-to-core file part>* requires more records than allowed by the *<core-to-core records per file part>*. [Note: The default is one record per file part.]
442	Hex string for single-precision list element contains more than 12 significant digits.
443	Unmatched quote for hex string, or non-hex character encountered in hex string.
444	Hex string for double-precision list element contains more than 24 significant digits.
462	Quoted string has not been exhausted, but next list element is a pointer (unresolvable ambiguity).
467	Input value exceeded the maximum value that the list element is capable of representing.
473	Unmatched quote.
484	An expression may not be used as a list element which receives a value on input.

The meanings of the various format error numbers pertaining to output are as follows:

NUMBER	ERROR MESSAGE
100	An error on output occurred when the intrinsic did a logical I/O.
102	Format was V specifier, and list element did not produce an A, C, D, E, F, G, H, I, J, K, L, O, P, R, S, T, U, X, or Z. [Note: If the list element is single precision, the rightmost character is used. If the list element is double precision, the rightmost character of the first (most significant) word is used. If the list element is a pointer, the character it points to is used.]
103	Format was V specifier of the form rV, and the resultant specifier needed a field width: e.g., 2V ⇒ 2I.
104	Format was V specifier of the form rV, and the resultant specifier needed a field width and decimal places: e.g., 2V ⇒ 2E.
105	Format was V specifier of the form rVw, and the resultant specifier needed decimal places: e.g., 2V* ⇒ 2F6.

NUMBER	ERROR MESSAGE
106	Format specifier evaluated to Fw.d form, and d<0.
107	Format specifier evaluated to Ew.d or Dw.d, and d<0.
109	Format specifier evaluated to Zw, and corresponding list element was neither of type integer nor type Boolean (expressions of type integer or Boolean are edited under Zw.d as Iw or Lw, respectively). Therefore, the decimal places are considered missing.
110	The list contains an element whose type is inappropriate for its associated format phrase. [Note that a pointer or a long (>48 bits) string cannot be used with a numeric editing phrase.]
111	Format specifier evaluated to Zw.d, and Zw.d logic chosen to edit the expression under Ew.d, but d<1.
113	Format specifier evaluated to Ew.d or Dw.d, and w≤d.
114	Dynamic w or d part of format specifier evaluated to a value greater than the maximum integer allowed, 549755813887.
116	Attempted recursive I/O – evaluation of a list element caused a read/write/close on the current file.
117	Record overflow – an attempt was made to output more characters than the record can have.
120	Output to the *<core-to-core file part>* requires more records than allowed by the *<core-to-core records per file part>*. [Note: The default is one record per file part.]
131	Dynamic r part of format specifier evaluated to a value greater than the maximum real allowed, 4.31359146673*10**68.
132	Dynamic w part of format specifier evaluated to a value greater than the maximum integer allowed, 549755813887.
133	Dynamic d part of format specifier evaluated to a value grearer than the maximum integer allowed, 549755813887.
163	Maxrecsize not large enough to allow freefield write.

The meanings of the various format error numbers pertaining to formatted input are as follows:

NUMBER	ERROR MESSAGE
200	An error on input occurred when the intrinsic did a logical I/O.

NUMBER	ERROR MESSAGE
202	Format was V specifier, and list element did not produce an A, C, D, E, F, G, H, I, J, K, L, O, P, R, S, T, X, or Z. [Note: If the list element is single precision, the rightmost character is used. If the list element is double precision, the rightmost character of the first (most significant) word is used. If the list element is a pointer, the character it points to is used.]
203	Format was V specifier of the form rV, and the resultant specifier needed a field width: e.g., 2V ⇒ 2I.
204	Format was V specifier of the form rV, and the resultant specifier needed a field width and decimal places: e.g., 2V ⇒ 2E.
205	Format was V specifier of the form rVw, and the resultant specifier needed decimal places: e.g., 2V* ⇒ 2F6.
206	Format specifier evaluated to Fw.d form, and d<0.
207	Format specifier evaluated to Ew.d or Dw.d, and d<0.
209	Format specifier evaluated to Zw, and corresponding list element was neither of type integer nor type Boolean (expressions of type integer or Boolean are edited under Zw.d as Iw or Lw, respectively). Therefore, the decimal places are considered missing.
210	The list contains an element whose type is inappropriate for its associated format phrase. [Note that a pointer or a long (>48 bits) string cannot be used with a numeric editing phrase.]
213	Format specifier evaluated to Ew.d or Dw.d, and w≤d.
214	Dynamic w or d part of format specifier evaluated to a value greater than the maximum integer allowed, 549755813887.
216	Attempted recursive I/O – evaluation of a list element caused a read/write/close on the current file.
217	Record overflow – an attempt was made to input more characters than the record has.
218	Invalid data for H or K format phrase.
220	Input from the *<core-to-core file part>* requires more records than allowed by the *<core-to-core records per file part>*. [Note: the default is one record per file part.]
231	Dynamic r part of format specifier evaluated to a value greater than the maximum real allowed, 4.31359146673*10**68.
232	Dynamic w part of format specifier evaluated to a value greater than the maximum integer allowed, 549755813887.

NUMBER	ERROR MESSAGE
233	Dynamic d part of format specifier evaluated to a value greater than the maximum integer allowed, 549755813887.
250	The U format phrase has yet to be implemented for input.
271	The $ and P format modifiers are not allowed on input.
281	Invalid data for I format phrase.
284	An expression as a list element which receives a value on input is not allowed.
285	The list element was type real, but the input value exceeded the maximum real allowed, 4.31359146673*10**68.
286	The list element was type integer or Boolean, but the input value exceeded the maximum integer allowed, 549755813887.
291	While inputting a constant using a numeric editing phrase, a non-digit was detected in the exponent part following at least one legitimate digit.
292	While inputting a constant using a numeric editing phrase, two or more exponent signs were detected.
293	While inputting a constant using a numeric editing phrase, an illegal character was detected after the exponent sign and before the exponent value.
294	While inputting a constant using a numeric editing phrase, an illegal character was detected past the decimal point.
295	While inputting a constant using a numeric editing phrase, two or more mantissa signs were detected.

APPENDIX H. COMPILE-TIME FACILITIES

INTRODUCTION

The compile-time facility is used conditionally and/or interactively to compile **ALGOL** source data, the following description consists of (1) the declaration and use of compile-time variables; (2) compile-time identifiers; (3) compile-time statements; and (4) dollar card options, the **ALGOL** compiler must be compiled with the option **CTPROC** set in order to include these features in compilations.

COMPILE-TIME VARIABLES

Syntax

<compile-time variable declaration> ::= **NUMBER** *<CT var list>*
<CT var list> ::= *<CT var>* | *<CT var list>* , *<CT var>*
<CT var> ::= *<identifier>* | *<identifier>* ::= *<initial value>* | *<identifier>* [*<vector length>*]
<initial value> ::= *<arithmetic expression>*
<vector length> ::= *<arithmetic expression>*

Semantics

An identifier declared to be a **NUMBER** is a "number variable", or an arithmetic compile-time variable. A number variable represents a single-precision arithmetic value. It may be used wherever an arithmetic value is allowed; it represents the value most recently assigned to it. The value of a number variable may be changed at any time during compilation by means of a *<compile-time 'LET' statement>*. A number variable may be declared with an *<initial value>*. By default, the *<initial value>* is zero. The *<initial value>* must be a constant *<arithmetic expression>*.

A vector of number variables may be declared by specifying its length in brackets. The members of a vector of number variables are referenced like subscripted variables. The subscript must be an arithmetic constant expression, greater than or equal to zero, and less than the declared vector length. The *<vector length>* must be a constant *<arithmetic expression>*.

COMPILE TIME IDENTIFIERS

Syntax

<compile-time identifier> ::= *<identifier>* *<apostrophe>* *<numberid>* |
<identifier> *<apostrophe>* *<compile-time variable>*

Semantics

A *<compile-time identifier>* is formed by combining a compile-time variable with an *<identifier>*. The *<compile-time identifier>* may appear anywhere a normal identifier may be used, including declarations. No blank characters may appear between the *<identifier>* and the *<apostrophe>*. The created *<identifier>* is the original *<identifier>* followed by an *<apostrophe>*, followed by numeric characters corresponding to the value of the compile-time variable, with leading zeros suppressed.

COMPILE-TIME STATEMENTS

Compile-time statements are introduced by the apostrophe. They are recognized at a very primitive level in the compiler and may, therefore, appear "almost anywhere," such as between any two normal language elements.

The compile-time statements are intended to provide a convenient method for altering the normal control of compilation, primarily via conditional and iterative compilation.

The compile-time statements (all introduced by an apostrophe) are as follows:

a. **BEGIN**
b. **IF**
c. **THRU**
d. **FOR**
e. **WHILE**
f. **DEFINE**
g. **INVOKE**
h. **LET**

In the syntax which follows, the term "text" refers to any **ALGOL** text, including complete compile-time statements.

Complete compile-time statements are always terminated by semicolons. However, compile-time statements which are components or other statements are terminated by **'END** or **'ELSE**. Note that these rules are the same as for normal **ALGOL** statements.

'BEING STATEMENT

Syntax

<compile-time begin> ::= **'BEGIN** *<text>* **'END** *<comments>*

Semantics

The *<compile-time begin stmt>* delimits a portion of **ALGOL** text. It is normally used in conjunction with the **'IF**, **'THRU** and **'FOR** statements. If the statement is not skipped, the **ALGOL** compiler processes all the delimited text; otherwise, the compiler ignores the text. Anything following the **'END** up to the first special character, **END**, **ELSE** or **UNTIL** is considered to be *<comments>* and is ignored.

'IF STATEMENT

Syntax

<compile-time if stmt> ::= **'IF** *<Boolean expression>* **THEN** *<ctstmt>* |
'IF *<Boolean expression>* **THEN** *<ctstmt>* **'ELSE** *<ctstmt>*

Semantics

This statement provides for conditional compilation of **ALGOL** text.

The *<Boolean expression>* must be a constant expression. If TRUE, the *<ctstmt>* following **THEN** is processed. If FALSE, the *<ctstmt>* following **'ELSE** is processed, if present.

'THRU STATEMENT

Syntax

<compile-time thru stmt> ::= **'THRU** *<arithmetic expression>* **DO** *<ctstmt>*

Semantics

The *<compile-time thru stmt>* provides iterative compilation of **ALGOL** text. The *<arithmetic expression>* must be a constant expression, greater than or equal to zero.

The *<ctstmt>* following **DO** is processed the specified number of times. If zero, the statement is skipped.

'FOR STATEMENT

Syntax

<compile-time for stmt> ::= **'FOR** *<numberid>* := *<aexp-1>* **STEP** *<aexp-2>* **UNTIL** *<aexp-3>*
DO *<ctstmt>*

Semantics

The *<compile-time for stmt>* provides for iterative compilation of **ALGOL** text. Each **AEXP** must be a constant *<arithmetic expression>*. *<aexp-2>* may be positive or negative, but must not be zero.

The action of this statement is similar to the **ALGOL** *<for statement>*. An exception is that *<aexp-2>* and *<aexp-3>* are evaluated only once, at the beginning of the operation. They are not re-evaluated, even though their components may change value.

'WHILE STATEMENT

Syntax

<compile-time while stmt> ::= **'WHILE** *<Boolean expression>* **DO** *<ctstmt>*

Semantics

The *<compile-time while stmt>* provides for iterative compilation of **ALGOL** text. The *<Boolean expression>* must be a constant expression. Prior to the iterations, the *<Boolean expression>* is evaluated. If it is **TRUE**, the *<ctstmt>* is processed; if it is **FALSE**, the entire statement is finished.

'DEFINE STATEMENT

Syntax

<compile-time define stmt> ::= **'DEFINE** *<identifier>* = *<ctstmt>*

Semantics

The *<compile-time define stmt>* declares an *<identifier>* to represent a *<ctstmt>*.

The *<ctstmt>* is stored away, to be processed when referenced in a subsequent *<compile-time invoke stmt>*.

'INVOKE STATEMENT

Syntax

<compile-time invoke stmt> ::= **'INVOKE** *<identifier>*

Semantics

The *<compile-time invoke stmt>* causes the *<ctstmt>*, previously associated with the *<identifier>* in a *<compile-time define stmt>* to be processed.

'LET STATEMENT

Syntax

<compile-time let stmt> ::= **'LET** *<number variable>* := *<aexp-2>*
<number variable> ::= *<numberid>* | *<numberid>* [aexp-2]

Semantics

This statement is used to modify the value of a compile-time variable. *<aexp-1>* must be a constant *<arithmetic expression>*. If the subscripted form is used, the *<numberid>* must have been declared as a vector of number variables; *<aexp-2>* must be greater than or equal to zero and less than the declared *<vector length>*.

COMPILER OPTIONS

- **CTMON**

  If **CTMON** is **SET**, all assignments to compile-time variables are monitored on the line printer listing. That is, the current value of a *<compile-time variable>* when it is referenced, and the new value when it is changed are printed.

- **CTLIST**

  If **CTLIST** is **SET**, all card images processed are printed on the line printer listing. In particular, during during iterative compile-time statements, card images which are processed repeatedly are printed repeatedly. They are identified by an asterisk (*) where the P or C usually appears. If **CTLIST** is **RESET** (default), card images are printed the first time they are processed.

- **LISTSKIP**

  If **LISTSKIP** is **RESET**, the printing of skipped card images is suppressed. If **LISTSKIP** is **SET** (default), card images are printed whether or not they are skipped (provided other listing options are set appropriately).

- **LISTINCL**

  If **LISTINCL** is **SET**, card images from the **INCLUDE** file are printed on the line printer listing (provided other listing options are set appropriately). If **LISTINCL** is **RESET** (default), included cards are not printed.

- **CTTRACE**

  If **CTTRACE** is **SET**, values of all expressions which are components of compile-time statements are printed on the line printer listing.

INDEX

In the page number column, the page number to the left of the semicolon refers to the text page that first uses or describes the metalinguistic item in the Extended . ALGOL syntax.

ITEM PAGE

ITEM PAGE

ITEM | PAGE

ITEM PAGE

ITEM	PAGE

ITEM PAGE

ITEM PAGE

ITEM PAGE

ITEM PAGE

ITEM PAGE

ITEM PAGE

ITEM PAGE

ITEM PAGE

ITEM PAGE

ITEM PAGE

ITEM	PAGE

ITEM PAGE

ITEM PAGE

ITEM PAGE